An Open-Source ML-Based Full-Stack Optimization Framework for Machine Learning Accelerators

HADI ESMAEILZADEH, University of California San Diego, La Jolla, United States
SOROUSH GHODRATI, University of California San Diego, La Jolla, United States
ANDREW KAHNG, CSE and ECE, University of California San Diego, La Jolla, United States
JOON KYUNG KIM, University of California San Diego, La Jolla, United States
SEAN KINZER, University of California San Diego, La Jolla, United States
SAYAK KUNDU, Electrical and Computer Engineering, University of California San Diego, La Jolla, United States
ROHAN MAHAPATRA, University of California San Diego, La Jolla, United States
SUSMITA DEY MANASI, University of Minnesota, Minneapolis, United States
SACHIN SAPATNEKAR, Electrical and Computer Engineering, Univ of Minnesota, Minneapolis, United States
ZHIANG WANG, ECE, University of California San Diego, La Jolla, United States
ZIQING ZENG, University of Minnesota, Minneapolis, United States

Parameterizable machine learning (ML) accelerators are the product of recent breakthroughs in ML. To fully enable their design space exploration (DSE), we propose a physical-design-driven, learning-based prediction framework for hardware-accelerated deep neural network (DNN) and non-DNN ML algorithms. It adopts a unified approach that combines power, performance, and area (PPA) analysis with frontend performance simulation, thereby achieving a realistic estimation of both backend PPA and system metrics such as runtime and energy. In addition, our framework includes a fully automated DSE technique, which optimizes backend and system metrics through an automated search of architectural and backend parameters. Experimental studies show that our approach consistently predicts backend PPA and system metrics with an average 7% or less prediction error for the ASIC implementation of two deep learning accelerator platforms, VTA and VeriGOOD-ML, in both a commercial 12 nm process and a research-oriented 45 nm process.

CCS Concepts: • **Hardware** → **Physical design (EDA)**; **Hardware accelerators**; **Application specific integrated circuits**.

Additional Key Words and Phrases: PPA prediction, design space exploration, ML accelerator

Authors' Contact Information: Hadi Esmaeilzadeh, University of California San Diego, La Jolla, California, United States; e-mail: hadi@eng.ucsd.edu; Soroush Ghodrati, University of California San Diego, La Jolla, California, United States; e-mail: soghodra@ucsd.edu; Andrew Kahng, CSE and ECE, University of California San Diego, La Jolla, California, United States; e-mail: abk@ucsd.edu; Joon Kyung Kim, University of California San Diego, La Jolla, California, United States; e-mail: jkkim@eng.ucsd.edu; Sean Kinzer, University of California San Diego, La Jolla, California, United States; e-mail: skinzer@eng.ucsd.edu; Sayak Kundu, Electrical and Computer Engineering, University of California San Diego, La Jolla, California, United States; e-mail: sakundu@ucsd.edu; Rohan Mahapatra, University of California San Diego, La Jolla, California, United States; e-mail: rohan@ucsd.edu; Susmita Dey Manasi, University of Minnesota, Minneapolis, Minnesota, United States; e-mail: manas018@umn.edu; Sachin Sapatnekar, Electrical and Computer Engineering, Univ of Minnesota, Minneapolis, Minnesota, United States; e-mail: sachin@umn.edu; Zhiang Wang, ECE, University of California San Diego, La Jolla, California, United States; e-mail: zhw033@ucsd.edu; Ziqing Zeng, University of Minnesota, Minneapolis, Minnesota, United States; e-mail: zeng0083@umn.edu.

ACM 1557-7309/2024/5-ART
https://doi.org/10.1145/3664652

1 INTRODUCTION

Recent advances in machine learning (ML) algorithms have catalyzed an increasing demand for application-specific ML hardware accelerators. The design of these accelerators is non-trivial: the design cycle from architecture to silicon implementation often takes months to years and involves a large team of cross-disciplinary experts. When faced with stringent time-to-market requirements, it is imperative to reduce turnaround time without sacrificing product quality.

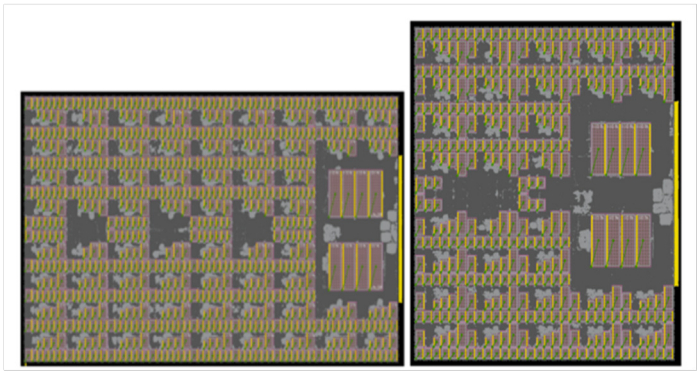
Recent research has developed automation flows for generating parameterizable accelerators, suitable for both FPGA and ASIC platforms. Parameterizable deep neural networks (DNNs) accelerators include VTA [3, 28], GeneSys [9], and Gemmini [12]. Accelerators for non-DNN ML algorithms [35], such as support vector machines or linear/logistic regression, have widespread applications, but have seen more limited research, with the TABLA platform [26] being a prominent example of a general-purpose non-DNN accelerator.

Generators such as those listed above allow designers to configure key parameters of DNN/non-DNN ML accelerators, e.g., the number of processing units or the on-chip memory configuration. The accelerator hardware description is then automatically translated to hardware at the register-transfer level (RTL). The search for an optimal configuration involves tradeoffs between the *power dissipation*, *performance*, and *area* (PPA) of the hardware platform, and the *energy* and *runtime* required to execute an ML algorithm on the platform. Therefore, this optimization involves the solution of two problems: (i) generating an ML accelerator that optimizes the PPA metrics; and (ii) selecting a PPA-optimized hardware configuration that optimizes system-level metrics such as the runtime and energy required to run an ML algorithm. Our work is the first to solve these two problems.

The prediction of *platform PPA* based on an architectural description is a longstanding challenge in electronic design automation. In modern nanoscale technologies, PPA is closely linked to physical design. Moreover, for many ML hardware platforms, a considerable fraction of the layout area is occupied by large memory macros whose presence exacerbates the problem of PPA prediction. The prediction of *system-level metrics*, such as the runtime and energy required to execute an ML algorithm, is performed using system-level simulation engines. These simulators model data transfer and computation within the accelerator to determine the number of operations, stall cycles, memory latencies, etc. Since they use the frequency and power metrics of the hardware platform as inputs, their accuracy depends on the quality of PPA prediction.

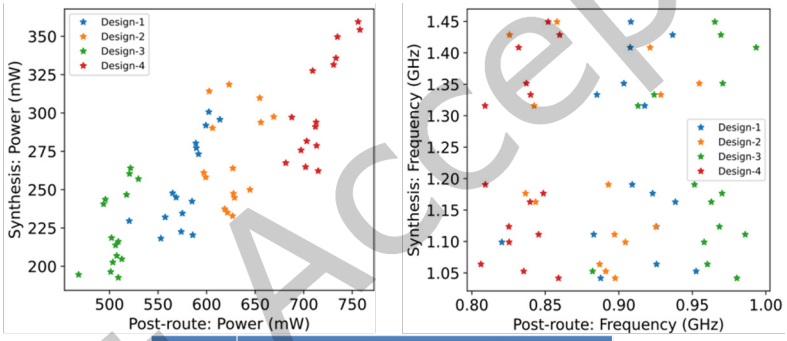
Given an ML accelerator, a target clock period, and a target floorplan utilization, the metrics of interest are the PPA of the hardware, and the energy and runtime required to execute ML algorithms. However, optimizing PPA does not necessarily guarantee optimal runtime and energy consumption. For instance, smaller hardware may consume less power but may not necessarily deliver the required improvements in energy consumption or meet runtime criteria. In Figure 1(a), which shows two designs implementing the same ML algorithm, we see that Design-B achieves 48% better power efficiency than Design-A. However, in terms of energy efficiency, it only shows 20% improvement and does so at a significantly slower runtime. This illustrates why Design Space Exploration (DSE) necessitates a rapid evaluator capable of assessing PPA, runtime, and energy consumption for numerous architectural configurations within a given design space. A DNN accelerator may easily have 5-10 million instances, and conventional evaluators require several days of synthesis, place and route (SP&R) runs to evaluate even a single configuration. Parallel evaluation runs cannot offer relief due to limited compute resources and available EDA tool licenses. Using post-synthesis PPA without P&R is inadequate: Figure 1(b) shows poor correlation between the post-synthesis and post-SP&R results for TABLA designs, visually and through the Kendall rank correlation coefficient (τ), where 0 signifies no correlation and ±1 indicates strong correlation or anti-correlation. For example, the τ values of four TABLA designs for total power are 0.61, −0.20, 0.07, 0.47, and for effective clock frequency are 0.45, −0.20, −0.16, 0.10.

To fully harness the potential of parameterizable ML accelerators, we propose a physical-design driven, learning-based prediction framework for hardware-accelerated ML algorithms. Our ML-based method accurately predicts



Design	PU	PE/ PU	Input bitwidth	Internal bitwidth	Area (mm^2)	Power (mW)	Frequency (GHz)	Runtime (µs)	Energy (µJ)
Design-A	8	8	16	32	2.96	580	0.95	**14.51**	8.41
Design-B	4	8	8	24	2.10	300	0.95	22.62	**6.80**

(a)



Design	Kendall Rank Correlation Coefficients (τ)	
	Power	Frequency
Design-1	0.60	-0.06
Design-2	0.09	-0.06
Design-3	0.35	-0.09
Design-4	0.54	0.14

(b)

Fig. 1. (a) Post-SP&R layouts of two TABLA designs implementing the same ML algorithm. Design-A achieves better runtime, while Design-B achieves better energy. (b) Miscorrelation of post-synthesis and post-routing metrics: total power and clock frequency for TABLA designs.

PPA and system performance, and overcomes the limitations associated with the high computational expense of design space exploration. We accomplish this by using a manageable number of SP&R backend runs to train ML models, which in turn predict the performance of ML accelerator designs for unseen configurations. Furthermore, we incorporate Multi-Objective Tree-structured Parzen Estimator (MOTPE)-based Bayesian optimization to automatically optimize the ML accelerators for the target ML algorithm and metric requirements, using the

trained models. Whereas previous works primarily focus on architectural design and/or RTL generation, our framework offers a full-stack optimization solution for ML accelerators, encompassing all aspects, from high-level ML algorithm specification to SP&R implementation guidelines. The main contributions of our work are as follows.

- We develop an ML-based full-stack optimization framework that integrates key design components ranging from ML algorithms and architectural parameters to SP&R recipes. This framework extends our previous work [10] to include chip area prediction and incorporates floorplan utilization as a backend feature, enhancing the accuracy of our prediction framework.
- We explore three sampling methods: Latin Hypercube sampling (LHS), and two low-discrepancy sequence (LDS) types, specifically Sobol and Halton sequences, across different sample sizes. Selecting the right sampling method and sample size is crucial for building high-accuracy prediction models. Additionally, the generation of train and test dataset using these sampling methods ensures uniform coverage of the design space, increasing the reliability of the ML model for DSE.
- We introduce a physical-design-driven, learning-based prediction methodology and a MOTPE-based method to automatically optimize ML accelerators for given target ML algorithm and metric requirements. Our experimental results suggest that our method can significantly reduce the implementation time of optimized ML accelerators, reducing both human effort and tool runtime from months to days.
- We present a novel method that capitalizes on the high degree of modularity present in ML accelerators. It generates a logical hierarchy graph, in which each leaf node represents a building block of the ML accelerators. This methodology employs a Graph Convolutional Network (GCN) to extract graph embeddings and train the model. Our experiment results indicate that the GCN model, even when trained on less data, can match or even outperform other models in predicting the test dataset.

In addition to the above, we have extensively tested our framework on two platforms, VTA and VeriGOOD-ML, and used different types of ML accelerators, including Non-DNNs (Axiline and TABLA), as well as DNNs (GeneSys and VTA). Among the non-DNN accelerators, Axiline is designed for smaller ML algorithms, while TABLA is intended for larger ML algorithms. In the category of DNN accelerators, GeneSys and VTA have been developed by two distinct research groups. The high accuracy and low error of our framework for these ML accelerators show that our framework is generalizable. We have made our SP&R flow scripts, model training, and DSE code publicly available in the VeriGOOD-ML GitHub repository [44].

The rest of this paper is organized as follows. Section 2 reviews the relevant literature on PPA prediction. Section 3 explains the key definitions and notations used in this paper. Section 4 presents the problem formulation for PPA prediction and design space exploration for ML accelerators. Section 5 provides an overview of our approach, with brief descriptions of each component of our proposed framework. Section 6 offers a detailed description of the logical hierarchy graph generation process used in GCN-based model training. Section 7 discusses the experimental setup, including data generation, data separation for model training and testing, and the details of the model training process. Section 8 presents our experimental results, which include an assessment of different sampling methods, the performance evaluation of our prediction models, and the results from DSE. Finally, Section 9 concludes the paper and outlines future research directions.

2 RELATED WORK

Prior efforts have sought to predict power, performance, and area at different stages of the design flow using two classes of predictors, respectively based on analytical models and ML models. The works in [15, 19] introduce ML models and demonstrate significant improvement in PPA prediction over previous analytical models such as ORION [37] and McPAT [21]. Works include power metric prediction for high-level synthesis (HLS) [8] and PPA prediction for memory compilers [18]. Such approaches indicate that ML models outperform analytical models

and are more convenient to train a wide range of ready-to-use ML models. To the best of our knowledge, no prior work builds ML models based on full backend SP&R for ASIC. In [2], an NN-based ML model is used to predict power and performance for different microarchitectures and to find Pareto optimal design points for power and performance. An ML model to estimate power metrics in HLS, along with sampling-based techniques to prune the search space, is presented in [24]; these methods are used to find the Pareto frontier and Pareto-optimal designs for FPGA implementations.

Recently, several studies have showcased the use of GNNs for predicting PPA, although they do not focus on ML hardware. The work in [33] directly predicts post-placement power and performance from RTL, demonstrating that the XGBoost model surpasses the performance of GCN-based models. The authors of [33] report 95% accuracy in terms of R-squared score, but calculation of the absolute percentage error from the provided data reveals a significant prediction error. Several other studies have employed GNNs for PPA prediction. For instance, [25] forecasts final PPA from the early stages of the P&R flow; [20] predicts post-synthesis PPA for Network-on-Chip (NoC) design using NoC parameters, topology configurations, and task graphs with the aid of a message-passing neural network; and [7] employs the graph representation of the netlist to predict leakage recovery during the ECO stage. Nevertheless, these studies do not offer prediction for post-route optimization or backend PPA from the RTL. In contrast, our work takes a novel approach for a given ML accelerator. We generate a logical hierarchy graph from the RTL and utilize GNNs to extract graph embeddings, which are then employed for backend PPA prediction.

In the ML hardware context, there is limited prior work on the early prediction of DNN accelerator performance. Aladdin [34] combines PPA-characterized building blocks with a dataflow graph representation to estimate performance, but does not incorporate the impact of physical design (PD) decisions beyond the block level. However, these decisions can substantially impact system performance. NeuPart [27] develops an analytical model to predict energy for computation and communication in a DNN accelerator. AutoDNNchip [39] proposes a predictor for energy, throughput, latency, and area overhead of DNN accelerators based on architectural parameters. It determines system-level performance metrics in an analytical-model-based coarse-grained mode and a runtime-simulation-based fine-grained mode, but has no clear engagement with backend design optimizations. On the other hand, it is well-understood that the performance of an ML accelerator is acutely dependent on the tradeoffs made in backend design. Numerous technology, methodology, and tool/flow effects must be comprehended, modeled and exploited – e.g., [1] shows that post-routing wirelength can be improved by about 15% with different settings of flow knobs. ML accelerators are considerably more complex than the testcases used in [1], and can be expected to show even greater overall variation in post-routing outcomes due to tool/flow effects. In contrast to all previous works, our approach takes the effect of backend flows into account and effectively ties the prediction of *system-level* performance metrics to backend PPA.

3 KEY DEFINITIONS AND NOTATION

We formally define a set of key terms used in our work.

- **ML accelerator (design).** An ML accelerator or design is the RTL netlist created by a parameterizable ML hardware generator.
- **Workload.** A workload is a user-specified ML algorithm (or a set of algorithms) that runs on an ML accelerator. Due to the inherent structure of the computation for a given network, the cost metrics for a workload – i.e., the energy and runtime of the accelerator – depend on the network topology and not on the specific input data.[1]

[1] The computational demand for a specific network is primarily determined by the network's structure and is largely independent of the input data values. This is illustrated in [6], which reports energy consumption for a single layer of AlexNet and VGG-16. Similarly, the work in [22] provides details on the energy usage for a single layer of AlexNet and GoogleNet-v1, giving additional confirmations that these measurements are not influenced by input data.

- **Architectural parameters.** These are a set of parameters used by a parameterizable ML hardware generator to generate an ML accelerator. An architectural configuration is a specific setting of architectural parameters. The parameterizable ML hardware generator can only generate one ML accelerator for a given configuration. This means there is a one-to-one mapping between ML accelerators and configurations for a parameterizable ML hardware generator.
- **Target clock period.** The target clock period is the clock period in the .sdc (Synopsys Design Constraints) file. The target clock frequency (f_{target}) is the reciprocal of target clock period. Altering f_{target} impacts the outcome of the SP&R process. SP&R tools aim to meet the specified f_{target}, neither more nor less. Exceeding the f_{target} can result in increased power consumption and area usage, whereas falling short of the f_{target} will lead to compromised performance.
- **Floorplan utilization.** Floorplan utilization ($util$) is an input parameter within the backend flow that determines the chip area based on the input synthesized netlist. The chip area is typically calculated as total standard cell and macro area within the synthesized netlist, divided by the floorplan utilization.
- **Backend parameters.** f_{target} and $util$ are the backend parameters. These parameters control the outcome of SP&R for a given ML accelerator RTL. A backend configuration is a specific setting of backend parameters.
- **Design configuration** is a specific setting for architectural and backend parameters.

For an ML accelerator:

- **Power (P)** is the sum of internal, switching, and leakage power, as reported by the SP&R tool after post-routing optimization.
- **Performance ($f_{effective}$)** of an ML accelerator is measured using the effective clock frequency ($f_{effective}$). The $f_{effective}$ indicates the maximum frequency at which the chip can operate. This is the reciprocal of effective clock period, defined as the target clock period minus the worst slack reported by the SP&R tool after post-routing optimization.
- **Area (A)** is the chip area reported by the P&R tool. For all our runs, we consider a chip aspect ratio of one, implying that the chips are square in shape.
- **Energy (E)** is the total energy required to run the user-specified workload on the ML accelerator. Given an ML accelerator and a workload, a simulator computes the energy based on the instruction mix and the post-SP&R performance/power metrics of the accelerator submodules.
- **Runtime (T)** is the time required to run the user-specified workload. For an ML accelerator and a specific workload, a performance simulator is used to compute the runtime.

4 PROBLEM FORMULATION

4.1 Prediction framework

We divide the problem of full-stack optimization of ML hardware accelerators into two subproblems. Given an ML architectural configuration $(x_1, x_2, \cdots, x_n)$, the logical hierarchy graph (LHG) of the RTL netlist, the target clock frequency and the floorplan utilization, two subproblems are as follows.

- **Problem 1:** predict power, performance and area for the backend implementation of a combination of architectural and backend configurations.
- **Problem 2:** predict system-level runtime and energy over a set of benchmark workloads.

To solve above two problems, we build two learning-based prediction frameworks:

- The first predicts power, performance and area

$$(P, f_{effective}, A) = MLModel(\{x_1, x_2, \cdots, x_n; LHG, f_{target}, util\}) \quad (1)$$

- The second predicts system-level runtime and energy

$$(T, E) = MLModel(\{x_1, x_2, \cdots, x_n; LHG, f_{target}, util\}) \tag{2}$$

For Problem 1, we train three separate models to predict each of power, performance and area. For Problem 2, we train two separate models to predict system-level runtime and energy.

We propose to leverage the RTL netlist produced by the ML hardware generator, together with architectural and backend features (f_{target}, $util$), as a strategy to enhance performance of our ML model. We apply the GCN modeling, which is adept at extracting relevant features from a graph. We generate an *LHG* from the RTL netlist, and incorporate this as an additional input into our GCN model, alongside the architectural and backend features.

4.2 Design Space Exploration

Next, we define the problem of design space exploration. Specifically, for a given target workload, our objective is to find the optimal architectural and backend configuration, which minimizes the cost function:

$$(E + \alpha \times A) \tag{3}$$

while satisfying following constraints.

- The total power (P) is less than the specified maximum power P_{max} ($P < P_{max}$).
- The runtime (R) is less than the specified maximum runtime R_{max} ($R < R_{max}$).
- The energy (E) and area (A) belongs to the Pareto front of (E, A) ((E, A) $\in$ *Pareto Front*(E, A)).

Here, α is a user-specified parameter and can be set to any positive real value. A higher value of α will prioritize area cost and a lower value will prioritize energy cost. In the experimental evaluation of our approach, when the energy is on the order of mJ and the chip area is on the order of mm^2, we set $\alpha = 1$ (as in VTA). Similarly, when the energy is on the order of μJ and the chip area is on the order of μm^2, we set $\alpha = 0.001$ (as in Axiline).

5 OVERVIEW OF OUR APPROACH

In this section, we first outline a broad overview of our framework. Figure 2 presents our overall framework, designed to optimize PPA and system-level metrics for ML accelerators. Here, color-coded boxes are used to indicate constant inputs, automated scripts, automated tool flows, and model training and cost optimization. Our framework consists of two parts, data generation and model training, followed by the use of the trained model for design space exploration. As shown in Figure 2, our overall framework works as follows. (1) We start by sampling architectural configurations for a given architectural and backend configurations space and generate RTL using RTL generators (see Subsection 5.1 for different ML accelerator platforms). (2) We then sample backend configurations, conduct SP&R and collect PPA information. We next run system-level simulation with the specified RTL and PPA information and capture the system-level metric data (Subsection 5.2 describes our sampling method). (3) We use the collected PPA and system-level metrics, along with the backend and architectural configurations, to train three backend ML models for PPA prediction and two system-level ML models for energy and runtime prediction. We use a two-stage model approach for ML model training and inference (see Subsection 5.3 for ML models and Subsection 5.4 for the two-stage model approach). (4) For a given workload, we use these trained models along with MOTPE to minimize energy and chip size while meeting power and runtime requirements (see Subsection 5.5 for DSE). (5) We identify the Pareto front for energy and area metrics, and apply Equation (3) to determine the optimal backend and architectural configurations.

In the following subsections, we cover platforms and simulators, introduce sampling methods, discuss various ML models, present the two-stage model, and outline our design space exploration strategy.

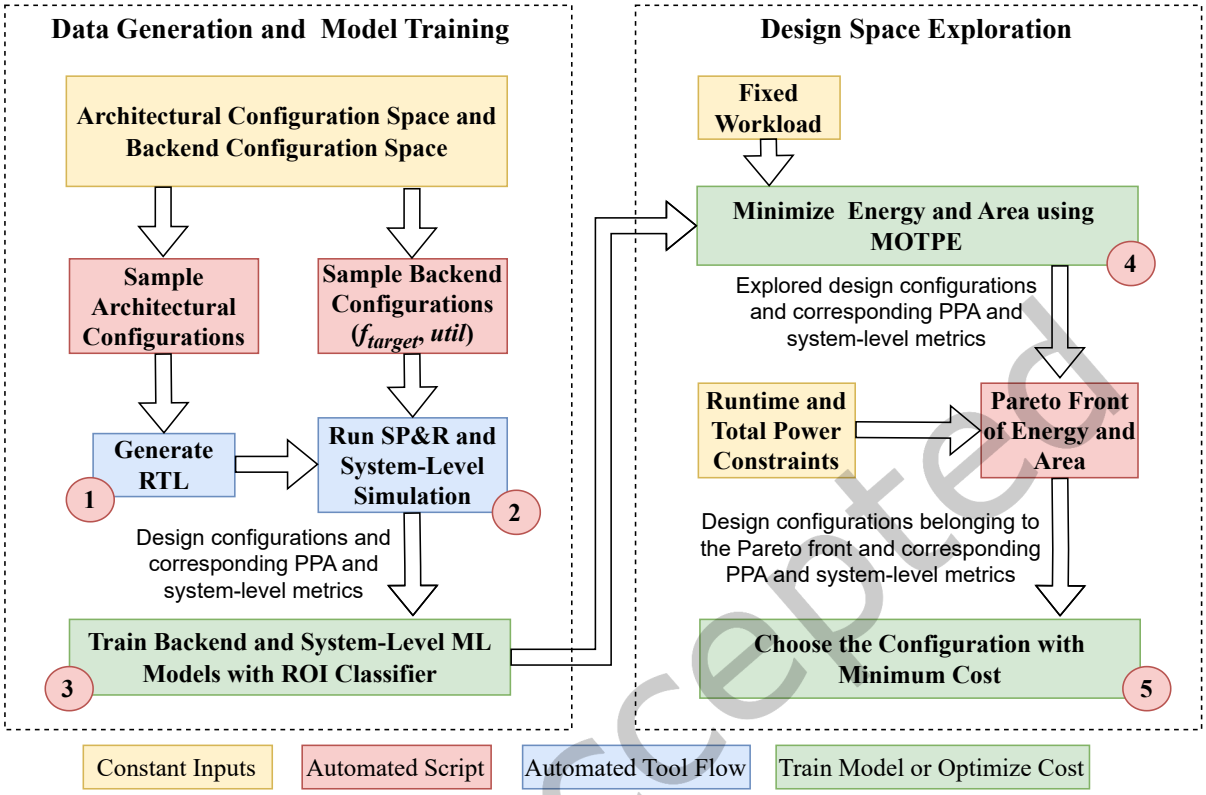


Fig. 2. Our ML-based full-stack optimization framework for machine learning accelerators.

5.1 Demonstration Platforms and Simulators

We demonstrate our approach on the accelerator engines from four parameterizable open-source ML hardware generators.

TABLA [26] implements non-DNN ML algorithms such as linear regression, logistic regression, support vector machines (SVMs), backpropagation, and recommender systems.

GeneSys [9] executes DNNs using an $M \times N$ systolic array for GEMM operations such as convolution, and an $N \times 1$ SIMD array for vector operations such as ReLU, pooling, and softmax.

Axiline [9, 40] builds hard-coded implementations of small ML algorithms (e.g., SVM, logistic/ linear regression) for training and inference.

VTA [3, 28] is a DNN accelerator where a compute module includes a GEMM core for convolution, along with a 2D array of PEs. VTA is integrated with Apache TVM [5], a deep learning compiler stack.

Integrating system simulations with backend data. The simulators are integrated with the backend analyses, where they receive PPA characteristics generated by our SP&R flow. The simulators used in our study are obtained from the GitHub repository of the VeriGOOD-ML project [41] and VTA hardware design stack [43]. For a specific hardware configuration point, provided as an input, the PPA characteristics feed the simulator with data such as the clock frequency, energy per access for each of the on-chip buffers, and dynamic and leakage power of

systolic and SIMD hardware components, or GEMM and ALU hardware components. The performance statistics provided by the simulator are combined with these backend data from the SP&R flow to produce end-to-end runtime, energy, and power for execution of a given ML algorithm.

5.2 Sampling Method

Sampling is a very important step for data generation. Improper sampling leads to an unbalanced dataset, resulting in poor performance of the trained models. Moreover, sampling techniques can help reduce the number of data points needed to train a model without degrading the performance. In this work, we have studied three sampling techniques, including (i) Latin Hypercube sampling, (ii) low-discrepancy sequence using Sobol and (iii) LDS using Halton.

Latin Hypercube sampling. LHS divides the parameter space into equally spaced intervals along each dimension and then randomly selects points in the intervals such that each interval is selected exactly once. During the sampling process we maximize the minimum pairwise distance of the sampled points.

LDS using Halton. An LDS, also known as a quasi-random sequence, is a sequence of points in a multi-dimensional space that exhibits more uniform and evenly distributed patterns, as compared to random sequences. These sequences are generated using deterministic algorithms that carefully distribute points across the space to minimize undesirable patterns. The Halton [30] sequence relies on unique prime number bases to generate uniformly distributed samples.

LDS using Sobol. The Sobol [30] sequence is also an LDS which utilizes primitive polynomials and bitwise operations to generated uniformly distributed samples. In the case of a high-dimensional parameter space, the Sobol sequence is expected to exhibit a more uniform distribution with a smaller sample size than the Halton sequence.

We use the scikit-optimize package [13] to generate samples for all three sampling methods. Both LHS and LDS yield superior results compared to random sampling, particularly when dealing with smaller sample sizes. One of the benefits of LHS is its ability to uniformly sample from the parameter space with a smaller sample size than LDS. However, a limitation of LHS is that to increase the number of samples, one needs to regenerate all the samples and cannot reuse the previously sampled data points. This is because adding new points to existing LHS samples disrupts the LHS property of maximizing the minimum pairwise distance between sampled points. On the other hand, LDS, which generates samples from a sequence, only needs to extend the sequence to generate new samples. This feature allows LDS to utilize previously sampled data points when the sample size is increased. In our experiments below (Section 7), we sample backend configuration $util$ and f_{target} for all designs, and architectural configuration size and number of cycles for Axiline designs.

5.3 ML Models

In our framework, we employ a range of regression models. A brief description of each model is as follows.

- **Gradient Boosted Decision Trees (GBDT)** utilize multiple decision trees as weak predictors. New trees are added sequentially to minimize the loss function during the training process.
- **Random Forest (RF)** also uses decision trees, but trains each tree independently using random samples of data. The final decision is generated based on voting over a set of trees or by averaging the predictions generated by each tree.
- **Artificial Neural Network (ANN)** is a biologically-inspired model consisting of multiple neuron or node layers: an input layer, one or more hidden layers, and an output layer with single node. The output of any node is a linear transformation of the outputs from the previous layer, followed by a non-linear or linear activation function. The single node in the output layer generates a scalar value corresponding to one target metric. We

train five different ANN models for predicting power, performance, area, system-level runtime, and energy, respectively.

- **Stacked Ensemble** uses multiple "base learner" algorithms to outperform each of the individual base learners. Training entails (i) training of multiple base learners (e.g., RF and GBDT models); and (ii) training of a second-level "meta learner" to find the optimal combination of the base learners as the stacked ensemble model. Theorem 1 in [17] proves that the stacked ensemble model will asymptotically perform as well as the best learner.
- **Graph Convolutional Network** performs convolution operations on graphs to capture structural dependencies and relationships inherent within the data. This is accomplished by iteratively propagating and aggregating information throughout the graph, enabling the model to learn from and adapt to the complex interconnected structure of the input data.

We train GBDT, RF and ANN models using the open-source platform H2O [42]. H2O enables fast, distributed, in-memory, and scalable machine learning-based model training. In Section 7.3, we provide a detailed step-by-step guide on how to train GBDT, RF, ANN and Stacked Ensemble models. We train GCN using PyTorch [32] and PyTorch Geometric [11]. In Section 6, we provide a detailed step-by-step guide on how to generate graphs, and in Section 7.3, we outline the process for training the models.

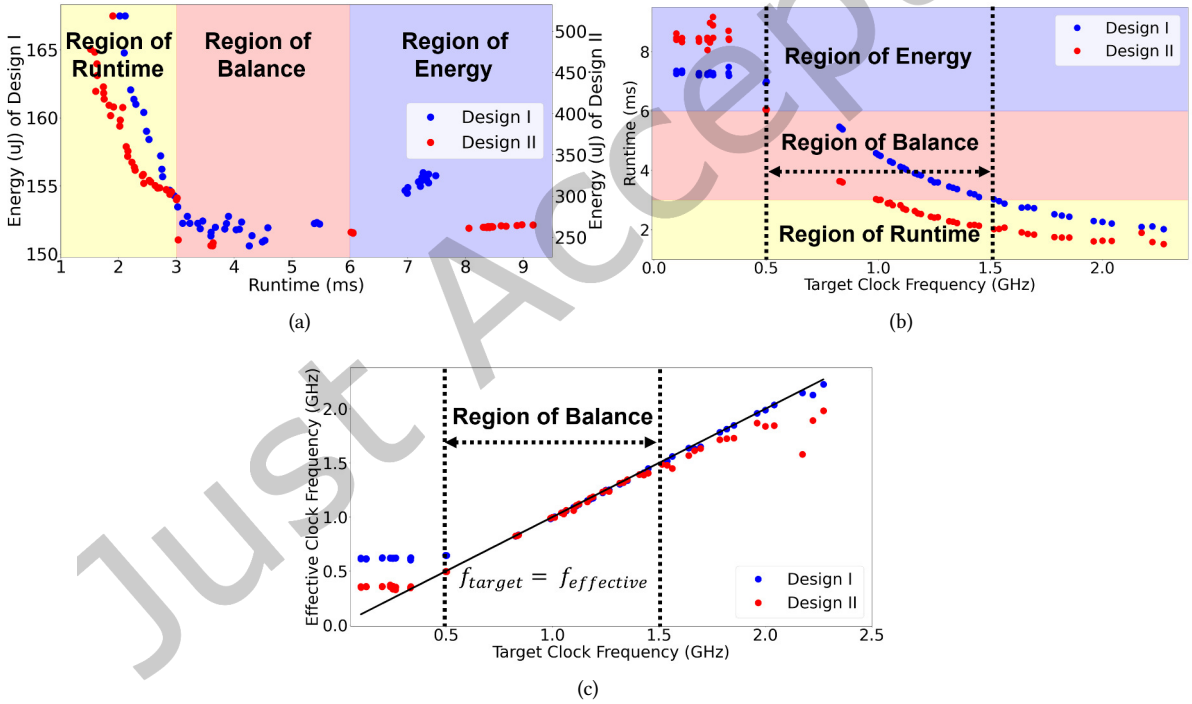


Fig. 3. Illustration of the region of interest (ROI). (a) Energy versus runtime. The ROI is the pink region. (b) Runtime versus target clock frequency. Again, the ROI is the pink region. (c) Target clock frequency versus effective clock frequency. The ROI is the region where $f_{target} = f_{effective}$. ROI is defined for each design individually. Here, the ROI for Design-I and Design-II happens to be exactly the same, although this is not always expected.

5.4 Two-Stage Model

Design space exploration seeks an optimal implementation of an accelerator for a specified workload, to achieve better backend PPA and system metrics. To explore the design space more efficiently, we pay more attention to a *region of interest* in the design space, i.e., the region that contains the "optimal" implementation. Figure 3 gives an example of how to determine the ROI. In the example, we use the Axiline platform to generate two accelerators (*Design-I* and *Design-II*) with different configurations, to implement a recommender system algorithm. Here the size, number of cycles, number of units, and bit width for *Design-I* are, respectively, 9, 7, 8, and 8, while for *Design-II* they are 17, 4, 5, and 16. We run full SP&R flows for these two accelerators under 21 different f_{target} values, and compute corresponding energy and runtime system metrics. Figure 3(a) shows energy versus runtime of *Design-I* and *Design-II* for different f_{target}. The data show a typical division of the design space into three regions: (i) region of *runtime* where we can reduce the runtime at the cost of increasing energy; (ii) region of *balance* where we can achieve lowest energy without introducing too much runtime overhead; and (iii) region of *energy* where the energy will also increase when the runtime increases. In our work, we set the ROI to be the region of balance, which is defined for each design. To visualize the ROI in terms of f_{target}, we show plots of runtime versus f_{target} in Figure 3(b).[2] We observe that the ROI excludes both extremely high and low f_{target}. Then, by examining $f_{effective}$ versus f_{target} (Figure 3(c)), we further characterize the ROI in terms of the difference between f_{target} and $f_{effective}$: post-routed designs tend to have smaller negative slacks at higher f_{target} and larger positive slacks at lower f_{target}. Given the above, we define our ROI in terms of the difference between f_{target} and $f_{effective}$, as follows:

$$ROI = \{f_{target} | abs(f_{effective} - f_{target}) \leq \epsilon \times f_{target}\} \tag{4}$$

The ROI for Axiline, VTA, and TABLA designs is identified by the data points situated between the red and blue lines in Figure 4. In Figure 4(a), for Axiline designs, we observe that for f_{target} values between 0.4 and 0.7, many data points lie above the red line, i.e., outside the ROI. This indicates that some of these data points fall on the left boundary of the ROI for different Axiline designs. Similarly, the right boundary of the ROI appears to range from 1.7 to 2.2 GHz. This exemplifies how the ROI can vary and may have different ranges for different designs.

We include floorplan utilization as a separate knob to control chip area. We observe that high utilization significantly impairs the outcomes of the backend P&R tool, leading to poor PPA. Figure 4(a) presents the $f_{effective}$ vs. f_{target} plot for the Axiline design on GF12. For f_{target} values of 1.03 and 1.33 GHz, the floorplan utilization hovers around 90%, which results in poor postRouteOpt performance for most of the Axiline designs. Additionally, when the f_{target} is very high, the P&R tool struggles to achieve the desired performance, resulting in a poor $f_{effective}$. Conversely, when f_{target} is very low, the tool yields a higher $f_{effective}$ than f_{target}. These extreme f_{target} values are also undesirable as the outcomes from the P&R tool for these frequencies tend to vary significantly, making them challenging to model accurately. So, these data points are considered as outliers, since they do not correspond to relevant design points, and since their inclusion in the training dataset deteriorates the performance of the model.

We modify the two-stage inference model proposed in [10] into a two-stage model based on the ROI definition above. The goal is to efficiently detect outliers and prevent them from impacting the performance of the model. As shown in Figure 5, the two-stage model functions as follows. (1) First, we generate classification labels for all training data points, each corresponding to a different design configuration, using Equation (4) to assess whether they fall within the ROI. (2) Next, we train a binary classification model to identify the data points that are within the ROI. (3) Subsequently, we train our models to predict PPA and system-level metrics, focusing only on the data points within the ROI. (4) During inference, we use the trained classification model to determine if the data points are within the ROI. (5) If the data points are within the ROI, we apply the trained backend and system-level models to predict PPA and system-level metrics; if not, these data points are excluded. In Equation (4), ϵ is a

[2] We emphasize that in Figure 3, we do not compare the ROI of *Design-I* and *Design-II* but only provide these as examples of ROI. Although Figure 3 shows identical ROI, such similarity is not typical. The ROI can vary between different designs.

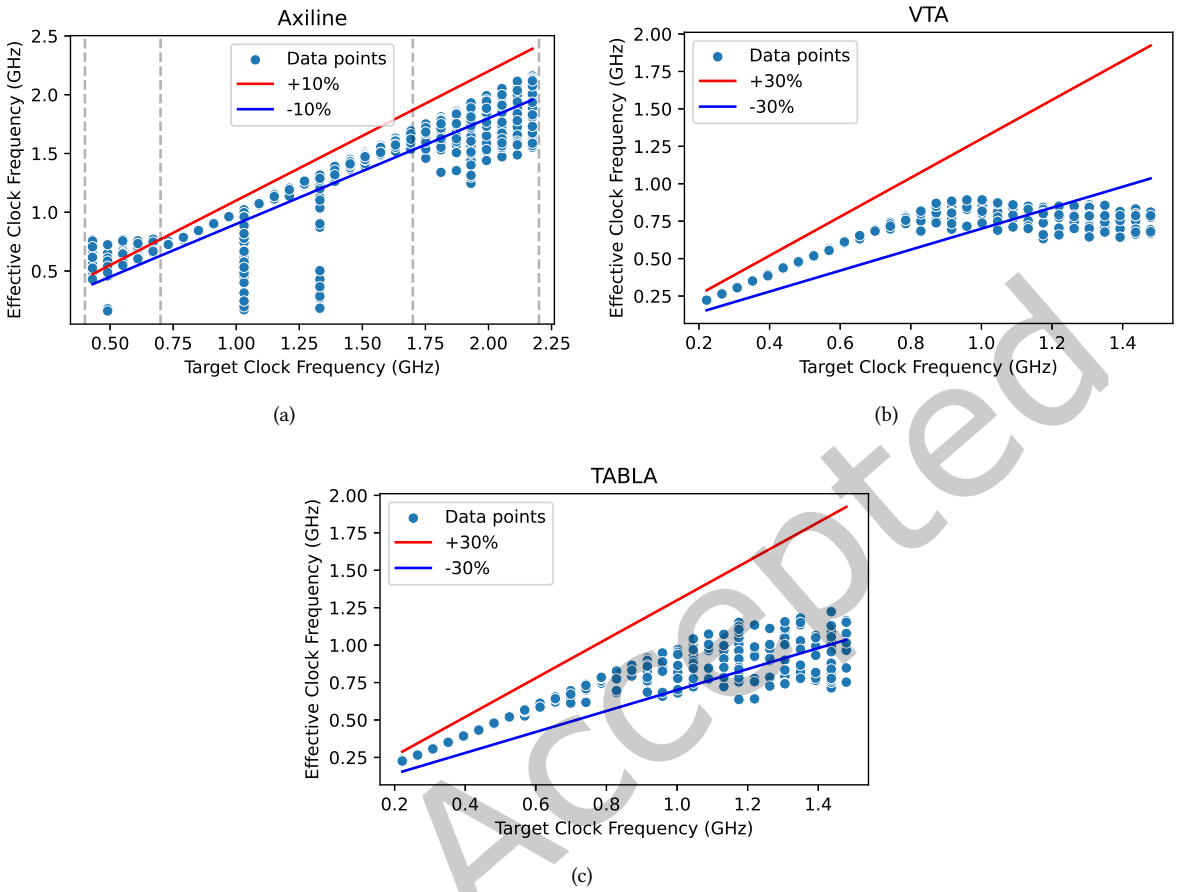


Fig. 4. Effective clock frequency vs. target clock frequency plot for (a) Axiline, (b) VTA and (c) TABLA designs on GF12 enablement. In this context, the floorplan utilization is not the same for all f_{target} values; it ranges from 0.4 to 0.9 for Axiline and 0.2 to 0.6 for TABLA, GeneSys and VTA. For more details, see Figure 7.

parameter used to define the size of the ROI. For smaller ML accelerators such as Axiline, where the deviation of the $f_{effective}$ from the f_{target} is generally small, we set ϵ to 0.1. For larger ML accelerators such as GeneSys, VTA and TABLA, where the deviation of $f_{effective}$ from f_{target} tends to be larger, we set ϵ to 0.3.

5.5 Design Space Exploration

In our methodology, we utilize the above two-stage model for DSE of both architectural and backend configurations. During this DSE process, for a given runtime and power constraints, our initial step involves the identification of the Pareto front for area and energy. This is accomplished via the MOTPE. Subsequently, the optimal configuration is selected according to a user-specified cost function.

MOTPE-based DSE. MOTPE [31] is a sequential model-based optimization technique. It iteratively builds a surrogate model to predict performance and gather additional data informed by this model. The estimator constructs distributions of good (G) and bad (B) samples, categorizing samples based on their relative position

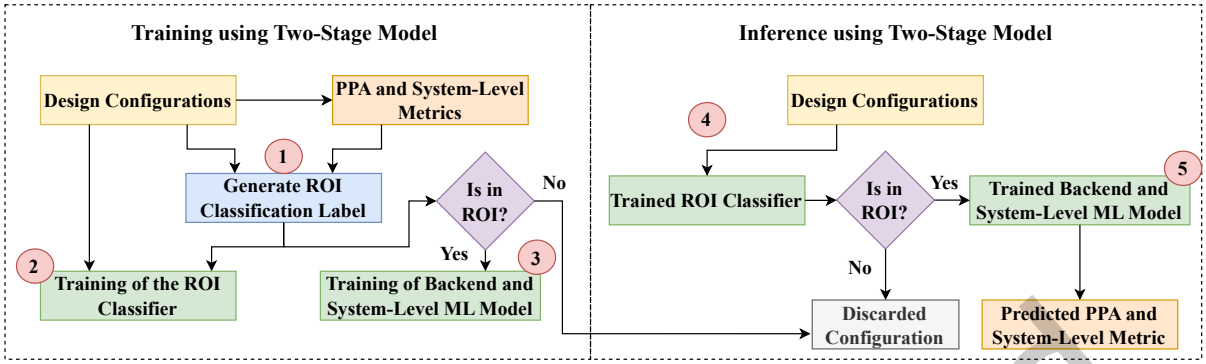


Fig. 5. Training and inference using the Two-Stage model.

in the objective space with respect to the current Pareto front. Before forming these internal distributions, the algorithm collects multiple random samples. It utilizes a non-parametric multivariate density estimation model, commonly referred to as the Parzen window, to construct these distributions. Subsequently, the MOTPE selects the sample which maximizes the G/B ratio, which ensures superior sampling efficiency. A significant advantage of MOTPE is its capacity to handle both discrete and continuous-valued parameters, thereby facilitating its application across a broad range of optimization scenarios. When conducting DSE of ML accelerators, backend parameters such as floorplan utilization and target clock frequency are continuous in nature, whereas architectural parameters such as the number of processing units and processing elements are discrete. This makes the MOTPE-based optimization well suited for our DSE problem. For a given workload, runtime, and power constraints, we use the MOTPE-based DSE approach to find the Pareto front for area and energy. Utilizing the cost function provided in Equation (3), we then select the best design configuration of the ML accelerator. Identifying the Pareto front allows us to understand the tradeoffs among different metrics, which is independent of user-specified weight in the cost function (i.e., α in Equation (3)).

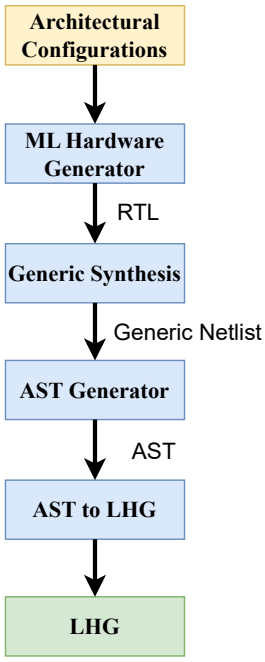
6 GRAPH GENERATION

ML accelerators exhibit a high degree of modularity in their design. For different architectural configurations, the ML hardware generators leverage various building blocks to generate the RTL netlist. We extract information about these building blocks and their connections from the RTL netlist. A graph representation of the connection between these building blocks is utilized in training our GCN models, which helps enhance model performance. For the GCN model, we extract the LHG from the RTL netlist. The specifics of the logical hierarchy graph are as follows.

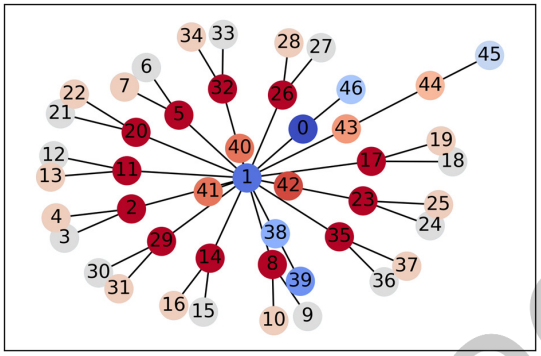
Logical hierarchy graph. As the name suggests, the LHG represents the logical hierarchy tree of the design for a given architectural configuration. Each module instantiation in the RTL netlist is mapped to a node in the LHG. Each undirected edge connects a parent module to its sub-module. All the leaf modules are the building blocks of the ML hardware generator. Figure 6(a) depicts the flow of generating an LHG for a given architectural configuration.

- We first generate the RTL netlist for a given architectural configuration, using an RTL generator such as VTA, GeneSys, TABLA or Axiline.
- Next, we transform the RTL netlist into a structural netlist, referred to as a *generic netlist*, using Cadence Genus 21.1.
- Using Pyverilog [36, 38], we generate the abstract syntax tree (AST) of the generic netlist.

- From the AST, we extract node features and generate the LHG using Algorithm 1.



(a)



(b)

Node Features	Details
#input, #output	count of input and output signals
#avg {in/out} bits	average number of input and output bits
#comb	number of combinational gates
#avg comb input bits	average number of inputs in comb gates
#flop	number of flip-flops
#memory	number of memory macros

(c)

Fig. 6. (a) Logical hierarchy graph generation flow. (b) Example of logical hierarchy graph of an Axiline design. Here, nodes with the same color correspond to the building blocks with the same functionality. (c) Details of node features.

The generic netlist serves as a structural representation of the RTL netlist, composed of Verilog primitives like *or*, *and* and *inv* for combinational logic, as well as dedicated modules such as flip-flops and latches for sequential logic. We generate and parse the AST of the generic netlist using Pyverilog [36, 38]. Algorithm 1 displays the detailed steps of LHG generation, as follows.

- Lines 1-2 generate a list of unique modules from the input AST, extracts features for each module and creates a *refNode* list. These features include the counts of input and output signals, the average number of input and output bits, the count of combinational cells, flip-flops, memories, and the average number of inputs per combinational cell.
- Line 3 updates the name of submodules, listing them as child nodes in the *refNode* list.
- Lines 4-6 initialize an empty graph, then the *AddNodeToGraph* procedure is used to create the LHG for a given *TopModuleName*.

The details of *AddNodeToGraph* procedure is as follows.

- Lines 1-5 add a node to the graph based on the input *refNode*. If the *refNode* is not the top module, the procedure also adds an edge to connect it to its parent node.
- Lines 6-8 add all the *subModuleNode* of the input *refNode* to the graph.

The advantage of using a generic netlist instead of the RTL netlist is that we can directly extract the number of combinational cells and the flip-flop count for each module, which is not possible with the AST generated

Algorithm 1: Generate Logical Hierarchy Graph from AST.

Input: AST, TopModuleName
Output: Logical hierarchy graph G

Parse AST using Pyverilog and create a list of unique modules
Extract features for each module and create a reference node list
Update the list of child nodes in the reference node list
$G \leftarrow$ Empty Graph; $id \leftarrow 0$; $pid \leftarrow -1$
AddNodeToGraph(*refNode* of TopModule, G, pid, id)
return G

Procedure: *AddNodeToGraph*
Input: refNode, G, pid, id
Output: id – is the node count of the updated graph G

$G.addNode(id, refNode)$
$node\_id \leftarrow id$; $id \leftarrow id + 1$
if $pid \neq -1$ **then**
  $G.addEdge(pid, node\_id)$
end
for $subModuleNode$ **in** $refNode.subModuleList$ **do**
  $id \leftarrow AddNodeToGraph(subModuleNode, G, node\_id, id)$
end
return id

directly from the RTL. During the synthesis process, the RTL netlist is mapped to a specific design library based on the given *Synopsys Design Constraints*. In contrast, during the generic netlist generation process, the RTL netlist is transformed to a structural format. So, the time taken to generate the generic netlist is significantly less than the synthesis runtime. For example, for a GeneSys design with 900K instances, the synthesis process takes 222 minutes, whereas the generic netlist generation process only takes 72 minutes.

Once we have all the node features, we instantiate the graph and utilize the *AddNodeToGraph* procedure to create the LHG. The *AddNodeToGraph* procedure initially adds the node corresponding to the top module, then uses depth-first search to add the node corresponding to the submodule. When adding the node corresponding to a submodule, it also creates an edge between the parent module and the submodule to ensure connectivity. Figure 6(b) shows the logical hierarchy graph of an Axiline design.

The number of nodes in an LHG, even for a very large ML accelerator with millions of instances, amounts to only a few thousand. For example, a GeneSys design with 900K instances has around 3,000 nodes in its LHG graph. The LHG graph represents the logical hierarchy tree, so the number of edges is one less than the number of nodes. Therefore, a high instance count does not pose any hindrance. Moreover, we opt to use graph convolutional layers such as *GCNConv* and *GraphConv*, over simple GNN layers such as *GraphSAGE*, because graph convolutional layers possess the capability to extract global features from the graph. Considering that the node count of the LHG is relatively low, on the order of thousands, the use of graph convolutional layers enables the efficient training of our models. We can thus navigate the large-scale structure of ML accelerators while maintaining a computationally tractable model. In Section 7.3, we include the details of the training process for the GCN model, and in Section 8, we discuss the performance of the model.

7 EXPERIMENTAL SETUP

In this section, we describe our experimental setup, which encompasses (i) the method of data generation; (ii) the separation of data into training and testing sets for the prediction of PPA along with system-level metrics for unseen backend and unseen architectural configurations; and (iii) the detailed steps involved in different model training procedures.

7.1 Data Generation

We divide the data generation process into three substeps: (i) sampling of architectural parameters and RTL netlist generation; (ii) sampling of backend parameters; and (iii) PPA and system-level metric data generation.

Sampling of architectural parameters and RTL netlist generation. All platforms, namely TABLA, GeneSys, VTA and Axiline, are parameterizable, with corresponding tunable architectural parameters shown in Table 1. We use a variety of strategies to generate multiple configurations for each platform. Prior works on DNN accelerators based on systolic arrays [14, 29] and vector dot-products [3] report architectural parameters such as array dimension, data bitwidth, on-chip buffer size and off-chip bandwidth. For GeneSys and VTA, we use such insights from prior works to guide our choice of architectural parameters. We proportionally scale buffer size and bandwidth parameters based on array dimensions. For each array dimension, we select other architectural parameters by changing ratios of the buffer sizes, so as to to exercise various data reuse tradeoffs in DNNs. For example, a larger WBUF facilitates more weight reuse while a larger OBUF biases a design toward more on-chip reduction of the partial sums. For TABLA, we explore multiple configurations using variations of the structures shown in [26]. For Axiline, we use *Latin Hypercube Sampling* for integer architectural parameters such as dimension and number of cycles, thus achieving uniform coverage in each dimension; and we simply enumerate all the combinations for all other remaining architectural parameters. In Section 8.1, we conduct a detailed analysis of the impact of various sampling methods and sample sizes on the prediction of unseen architectural configurations, specifically for the Axiline design. In Section 8.3, we show that our model underperforms when the testing dataset falls outside the range of the training dataset. Therefore, it is essential to ensure that the range of training dataset covers the testing dataset.

Sampling of backend parameters. As mentioned in Section 4, we incorporate the floorplan utilization as another backend parameter alongside the target clock frequency. This allows for the prediction of chip area. If we were to separately sample the target clock period and floorplan utilization, and then enumerate all configurations, it would significantly inflate the total number of configurations and be highly inefficient. Thus, we employ LHS to sample backend configurations. The performance of a chip is directly proportional to the clock frequency, while it is inversely proportional to the clock period. So, we sample from the target clock frequency space during backend parameter sampling. We then convert the frequency to the target clock period for the SP&R data generation. Figures 7(a) and (b) show the backend configurations sampled for Axiline, and for TABLA, GeneSys and VTA designs. We sample 30 backend configurations (f_{target}, $util$) for training and 10 for testing. We conduct f_{target} sweeps and set the f_{target} range to ensure that the worst slack is less than 200 ps and more than -200 ps. For the $util$ sweep, we set the upper boundary at the highest floorplan utilization that leads to a DRC count less than 100, and set the lower boundary at a difference of 0.4 to 0.5 from the upper boundary. Figure 4 presents how $f_{effective}$ varies with f_{target} for Axiline, TABLA and VTA designs. Based on the results of these sweeps, for Axiline, we sample floorplan utilization from 40% to 90% and target clock frequency from 0.4 GHz to 2.2 GHz. For the macro-heavy TABLA, GeneSys and VTA designs, we sample floorplan utilization from 20% to 60% and target clock frequency from 0.2 GHz to 1.5 GHz.

PPA and system-level metric data generation. After sampling the architectural configurations, generating the RTL, and sampling the backend configurations, we run commercial synthesis, place and route, and collect

[3]The benchmark parameter, used by TABLA and Axiline to decide the target ML algorithm, is not an architectural parameter.

Table 1. Architectural parameters and benchmarks for four design platforms.

Platforms	Feature	Candidate Values	Description
TABLA	PU	4, 8	# processing units
	PE	8, 16	# processing engines in each PU
	bitwidth	8, 16	bit width of internal bus
	input bitwidth	16, 32	bit width of *IO* bus
	benchmark[3]	recommender systems backpropagation	ML algorithms
GeneSys	weight data width	4 – 8 (integer)	bit width of weight data (bit)
	activation data width	4 – 8 (integer)	bit width of input activation data (bit)
	accumulation width	32 (integer)	bit width of output accumulation (bit)
	WBUF capacity	16 – 256 (integer)	size of weight buffer (KB)
	IBUF capacity	16 – 128 (integer)	size of input buffer (KB)
	OBUF capacity	128 – 1024 (integer)	size of output buffer (KB)
	SIMD VMEM capacity	128 – 1024 (integer)	size of vector memory in VMEM (KB)
	WBUF AXI data width	64 – 256 (integer)	AXI bandwidth for the WBUF (bits/cycle)
	IBUF AXI data width	128 – 256 (integer)	AXI bandwidth for the IBUF (bits/cycle)
	OBUF AXI data width	128 – 256 (integer)	AXI bandwidth for the OBUF (bits/cycle)
	SIMD AXI data width	128 – 256 (integer)	AXI bandwidth for the VMEM (bits/cycle)
VTA	weight data width	8 (integer)	bit width of weight data (bit)
	activation data width	8 (integer)	bit width of input activation data (bit)
	accumulation width	32 (integer)	bit width of output accumulation (bit)
	WBUF capacity	16 – 256 (integer)	size of weight buffer (KB)
	IBUF capacity	16 – 128 (integer)	size of input buffer (KB)
	OBUF capacity	32 – 512 (integer)	size of output buffer (KB)
	off-chip bandwidth	64 – 512 (integer)	total external bandwidth (bits/cycle)
Axiline	benchmark[3]	SVM, linear regression, logistic regression, recommender systems	ML algorithms
	bitwidth	8, 16	bit width for computation units
	input bitwidth	4, 8	bit width for initial inputs
	size	5 – 60 (integer)	dimension of inner product stage or SGD stage (both are the same)
	num of cycles	1 – 25 (integer)	number of cycles required for stages 1 or 3 to process one input vector

post-route optimization PPA metrics. We execute the logic synthesis using Synopsys Design Compiler R-2020.09, generating the synthesized netlist. We then carry out place and route using Cadence Innovus 21.1 to capture the post-route optimization PPA metrics. All of our tool scripts are available in the VeriGood-ML GitHub repository [44]. For macro-heavy designs, we employ Innovus' concurrent macro placer to automatically place all the macros. We conduct all of our studies using the GLOBALFOUNDRIES 12LP (GF12) enablement. For Axiline, we also generate PPA metrics on the open enablement NanGate45 [45] to demonstrate the adaptability of our framework across different platforms. Once we have the post-route optimization database, we run simulations to capture the system-level runtime and energy metrics. Here, we set the workloads to be the widely-used

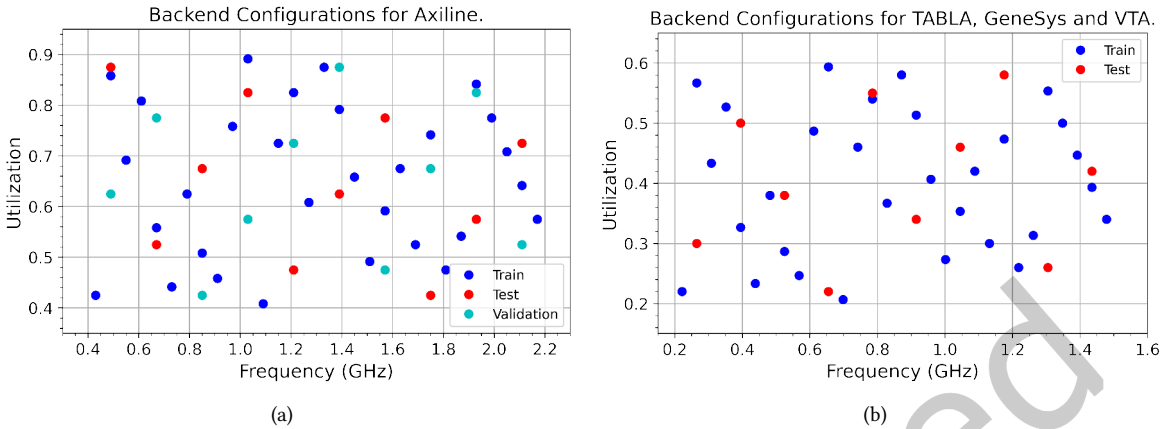


Fig. 7. Backend configurations sampled using Latin hypercube sampling. Blue dots correspond to the training dataset and red dots correspond to the testing dataset.

ResNet-50 and MobileNet-v1 networks for the GeneSys and VTA designs, respectively. The workload of each TABLA or Axiline design is determined by its benchmark parameter (see Table 1). For the training and testing of the ML model, we generate around 9800 data points for Axiline across both NG45 and GF12 enablements, and approximately 400, 320, and 190 data points for TABLA, VTA, and GeneSys, respectively.

7.2 Dataset Separation

After data generation, we divide the data into training and testing datasets. Our model is trained and tested using two distinct types of datasets, categorized based on backend and architectural configurations. These are referred to as the *unseen backend dataset* and the *unseen architectural dataset*. In the subsequent subsections, we first explore the significance of having a separate validation dataset rather than solely depending on cross-validation. We then delve into the details of the *unseen backend dataset* and the *unseen architectural dataset*.

Preference for validation dataset over cross-validation. We find that one of the primary causes of the poor performance of the ANN model in our previous study [10] is related to the use of cross-validation. Using cross-validation does not necessarily ensure adequate coverage of the design space, despite the fact that the complete training dataset has this property. This leads to a bias in the model that develops during the training process, which results in high μAPE and MAPE. As mentioned in Section 7.2, by separately sampling a validation dataset that provides a thorough coverage of the design space, we achieve a substantial improvement in the performance of the model on the testing dataset. During hyperparameter tuning, we employ the validation dataset to select the best hyperparameters and then deploy them on the testing dataset for evaluation. We apply this approach to the model training of Axiline designs. However, for TABLA, GeneSys, and VTA designs, generating the RTL netlist for different architectural configurations requires significant manual effort. Although using a validation dataset offers additional benefits, due to the substantial manual effort required, we have chosen to employ five-fold cross-validation for these designs.

Unseen backend dataset. As the name implies, the training and testing datasets encompass entirely different sets of backend configurations, though the architectural configurations within both datasets remain the same. We sample 30 data points for training and 10 for testing, making sure there is no overlap between any of these datasets. This distinct sampling helps to ensure coverage of the entire backend design space. Figure 7 displays

the backend configurations sampled for Axiline, TABLA, GeneSys and VTA for training, and testing with GF12 enablement. For Axiline, an additional 10 data points are sampled for model validation during the training process.
Unseen architectural dataset. In this case, the training and testing datasets comprise entirely of different sets of architectural configurations, while the backend configurations remain consistent across both datasets. Just as with unseen backend configurations, we ensure no overlap between the training and testing datasets. For the Axiline design, the training dataset contains 24 configurations, while the validation and testing datasets each consist of 10 configurations. Again, we separately sample all three (i.e., training, validation, and testing) datasets using LHS to help ensure a coverage of each dataset over the architectural design space. Section 8.1 discusses in detail the impact of sampling training configurations using different methods for varying sample sizes.

For Axiline, architectural configurations are sampled separately for training and testing, ensuring that both datasets uniformly cover the design space. Effective performance of the trained model on the test dataset suggests that the model will be reliable during DSE. However, for TABLA, GeneSys, and VTA, the substantial manual effort required to generate RTL necessitates a random division of the dataset into a 4:1 training-to-testing ratio, based on architectural configurations.

7.3 Model Training

In this work, we train and evaluate a total of 200 ML models, spanning four platforms (Axiline, TABLA, GeneSys and VTA), five metrics (power, performance, area, energy, and runtime), five types of machine learning models (GBDT, RF, ANN, Stacked Ensemble and GCN) and two datasets (unseen backend and unseen architectural). We tune the hyperparameters of each model except GCN, using the H2O package and a random discrete search method. Table 2 presents the hyperparameters of each machine learning model type that we tune, and for other parameters, we use the default values unless mentioned otherwise. In the following subsections, we delve into the detailed steps of hyperparameter tuning for each model.
Training of GBDT and RF. We use H2O and adopt the same strategy to train both the GBDT and RF models. Hyperparameter tuning using H2O random discrete [4] grid search is executed in two stages. In the first stage, we set the number of trees to a very large value (500 for RF and 300 for XGB) and tune the remaining hyperparameters. From the best hyperparameter configuration, we find the best *max_depth* to narrow down the search space for *max_depth*. For RF, we also reduce the search space for *mtries*. The range for *max_depth* is reduced to best *max_depth* ± 3. For RF, we retain the *mtries* value determined in the first stage. In the second stage, we conduct another round of hyperparameter search with the updated search space for *max_depth* in both RF and GBDT and for *mtries* in RF. During the random discrete search process, we choose the model with the lowest Root Mean Square Error (RMSE), computed using Equation (5) on the validation dataset of size n. In this equation, y_{actual} represents the actual value of the target metric, and $y_{predicted}$ is the value predicted by the trained model. This optimized model is then applied to the testing dataset. During hyperparameter tuning with random discrete grid search, we set the wall time to 300 seconds for each stage. Therefore, the overall model training time of the GBDT and RF models is limited to 600 seconds or less.

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_{actual} - y_{predicted})^2} \tag{5}$$

Training of ANN. For the ANN model, determining an effective hidden layer configuration is crucial. In previous work [10], hidden layer configurations are generated such that all hidden layers have identical node configurations, which leaves further room for improvement. The key idea in our present work is to map the features to a higher dimensional space and then gradually reduce them to a smaller dimension. This allows the model to extract complex representations and make the data more separable. To achieve this, we apply Algorithm 2 to create the hidden layer configurations. In Algorithm 2, *nodeCount* represents the count of nodes in the first hidden layer,

Table 2. Tuned hyperparameters for *GBDT*, *RF* and *ANN*.

Model	Parameters	Type	Range	Description
GBDT	n_estimator	integer	[20 – 500]	# gradient boosted trees
	max_depth	integer	[2 – 20]	maximum tree depth
RF	n_estimator	integer	[50 – 1000]	# decision trees in the forest
	mtries	*enum*	[1 – total feature count]	# features considered for best split
	max_depth	integer	[5 – 100]	max tree depth
ANN	num_layer	integer	[3 – 9]	# hidden layers
	num_node	enum	[8, 16, 32]	*nodeCount* input in Algorithm 2
	act_func	*enum*	[Tanh, Rectifier, Maxout]	activation function
GCN	conv_layer	*enum*	[GraphConv, GCNConv]	type of graph convolutional layer
	num_conv_layer	integer	[2 – 6]	# convolutional layers
	num_fc_layer	integer	[2 – 9]	# fully connected layers
	batch_size	integer	[16, 32, 64]	training batch size
	lr	float	$[10^{-2} - 10^{-5}]$	learning rate

Algorithm 2: Generation of hidden layer configurations.

Input: nodeCount, hLayerCount, minP = 2, maxP = 7
Output: $layer - i^{th}$ element of the list is the number of nodes in the i^{th} hidden layer.

1 **Function** `getNodeConfig`(*nodeCount, hLayerCount, minP, maxP*)**:**
2 $P = ceil(log_2(nodeCount))$;
3 $expMaxP = min((hLayerCount + minP + P)/2, maxP)$;
4 **if** $expMaxP \leq P$ **then**
5 $expMaxP = P + 1$;
6 **end**
7 $incrP = expMaxP - P$;
8 $decrP = min(expMaxP - minP + 1, hLayerCount - incrP)$;
9 $sameP = 0$;
10 **if** $hLayerCount > incrP + decrP$ **then**
11 $sameP = hLayerCount - incrP - decrP$;
12 **end**
13 $layer = []$;
14 Add 2^P to *layer* $incrP$ times while increasing P by 1;
15 Add 2^P to *layer* $sameP$ times;
16 Add 2^P to *layer* $decrP$ times while decreasing P by 1;
17 **return** *layer*;

and *hLayerCount* denotes the total number of hidden layers. The values *minP* and *maxP* are used to constrain the minimum and maximum number of nodes in the hidden layer, respectively, to 2^{minP} and 2^{maxP}.

- Lines 2-3 calculate *expMaxP*. The value $2^{expMaxP}$ represents the expected maximum number of nodes in the hidden layer.

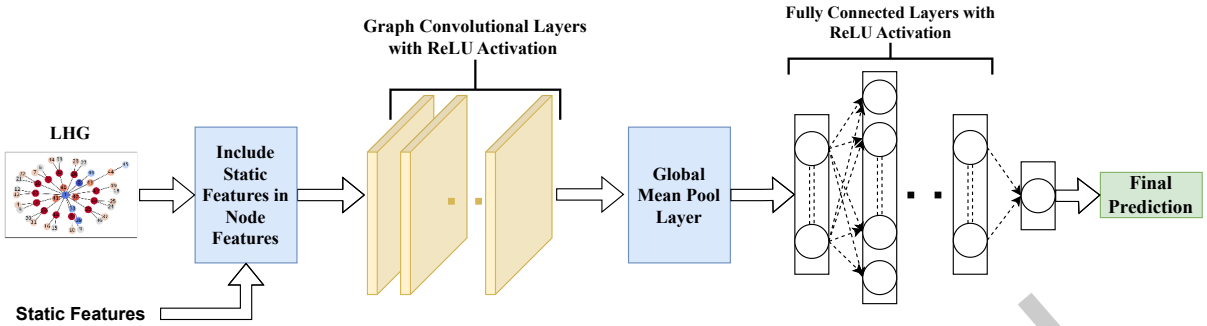


Fig. 8. Configuration of the GCN architecture used for backend PPA and system-metric prediction.

- Lines 4-6 update *expMaxP* if the expected maximum node count is fewer than the number of input nodes.
- Lines 7-12 determine respectively the number of layers where the node count will increase, decrease, and remain constant at $2^{expMaxP}$.
- Lines 13-17 calculate the node count for each layer and return the layer configuration.

Our algorithm consistently uses a power of two for the node count in each hidden layer, which is more resource-efficient. We employ the parameter values provided in Table 2 to generate all the configurations for the hidden layers. We perform the hyperparameter search for these hidden layer configurations and the set of activation functions presented in Table 2. During model training, we leverage adaptive learning rates. The model exhibiting the lowest RMSE value (Equation (5)) on the validation dataset is selected. We carry out this hyperparameter tuning of ANN models using the H2O random discrete [4] grid search. The selected model is then applied to the testing dataset for evaluation. During hyperparameter tuning, we use the default batch size of 1 and set the number of epochs to 500. The size of the training dataset is on the order of several thousands, so employing a batch size of 1 does not slow down the training process. We implement an early stopping criterion: if there is no improvement in the validation dataset for 20 consecutive iterations, the training process is stopped. During the hyperparameter search, the wall time is set to 600 seconds. Therefore, the overall training time of the ANN model is limited to 600 seconds or less.

Training of Stacked Ensemble. We employ the stacked ensemble model of H2O, where we use the trained models of GBDT, RF and ANN as base learners, with linear regression acting as meta learner. Here, only the top seven models from the hyperparameter search process of GBDT, RF and ANN are chosen as the base learners.[4] This approach ensures a sufficient degree of variation in the selection of base learners, while filtering out the poorly performing models.

Training of GCN. We have implemented the GCN model using PyTorch and PyTorch Geometric. The GCN architecture used to train our model is depicted in Figure 8. For both the graph convolutional layers and the fully connected layers, we use the *ReLU* activation function. The convolutional layers generate node embeddings, and we then employ the *GlobalMeanPool* function that computes the average of all the node embeddings, i.e.,

$$GlobalMeanPool(\mathbf{X}) = \frac{1}{N}\sum_{i=1}^{N}\mathbf{X}_i \tag{6}$$

The GCN models incorporate not only architectural and backend features but also extract features from *LHG*s. Given that ML accelerators are inherently modular and have shared building blocks, this enables the GCN models

[4]Our background studies empirically suggest that choosing the top seven models gives the best results.

to produce more insightful embeddings for PPA and system-level metric predictions. Figure 9 shows the t-SNE plot of the graph embeddings generated for TABLA, VTA, and Axiline designs. Here, different colors are used to plot graph embeddings of different architectural configurations. In the t-SNE plot, we see clear distinctions between different architectural configurations. Since the PPA and system-level metrics differ for these configurations, it indicates that the GCN models are well-trained.

Following the generation of graph embeddings, we input them into the fully connected layer to produce the final predictions. The configuration of the fully connected layer is generated using the *getNodeConfig* function, with the graph embedding size and the number of fully connected layers serving as inputs, corresponding to *nodeCount* and *hLayerCount*, respectively. While training our model, we employ the mean absolute percentage error (μAPE) loss, defined as

$$\mu APE = \frac{1}{n}\sum_{i=1}^{n}\left|\frac{y_{actual} - y_{predicted}}{y_{actual}}\right| \times 100 \tag{7}$$

Throughout the training of the GCN model, we utilize the Adam [16] optimizer with decaying learning rate; decaying factor is set to 0.7 with a patience of 5. We also implement early stopping if no improvement in the model's performance on the validation dataset is observed for 20 consecutive epochs. Upon completion of the training, the model exhibiting the lowest validation error is chosen for testing on the test dataset.

We employ the *HyperOptSearch* function of Ray Tune [23] to automatically optimize various parameters such as the type of graph convolutional layer, the number of convolutional layers, the number of fully connected layers, the batch size, and the learning rate as shown in Table 2. The best configuration is determined based on the following loss function,[5] which captures both the average and worst-case performance:

$$loss = \mu APE + 0.3 \times MAPE \tag{8}$$

During the training of our GCN model, we set the number of epochs to 400. However, most of the runs complete earlier due to the early stopping mechanism. We have observed that with 100 samples, the total runtime of Ray Tune is approximately 1800 seconds.

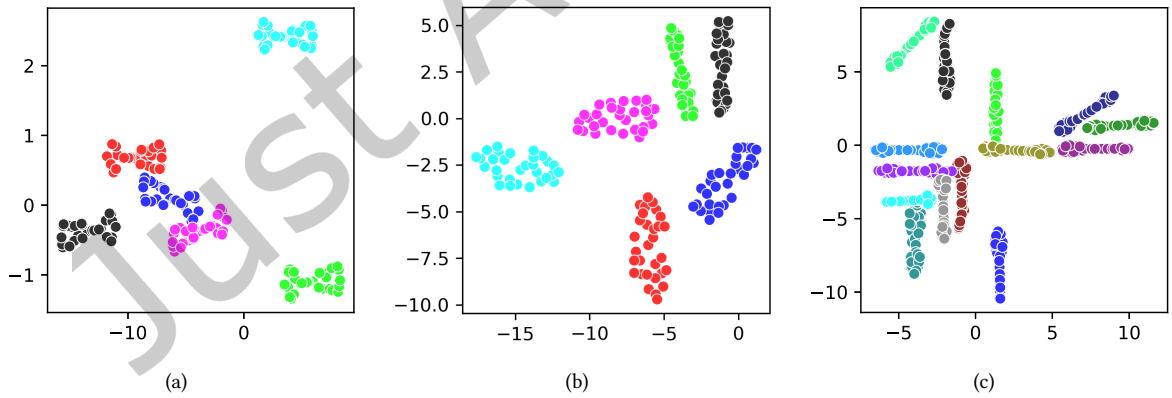


Fig. 9. t-SNE plot of the graph embedding generated by the trained GCN models for (a) TABLA, (b) VTA and (c) Axiline designs. Here, the data points highlighted with same color represent the same architectural configuration, but with different backend configurations.

[5]In our experiments with the GCN model, the value of μAPE varies within the range [0 – 20], whereas $MAPE$ varies in the range [0 – 60]. To ensure both μAPE and $MAPE$ on the same scale, we use a weight of 0.3 ($20/60 \approx 0.3$) for $MAPE$.

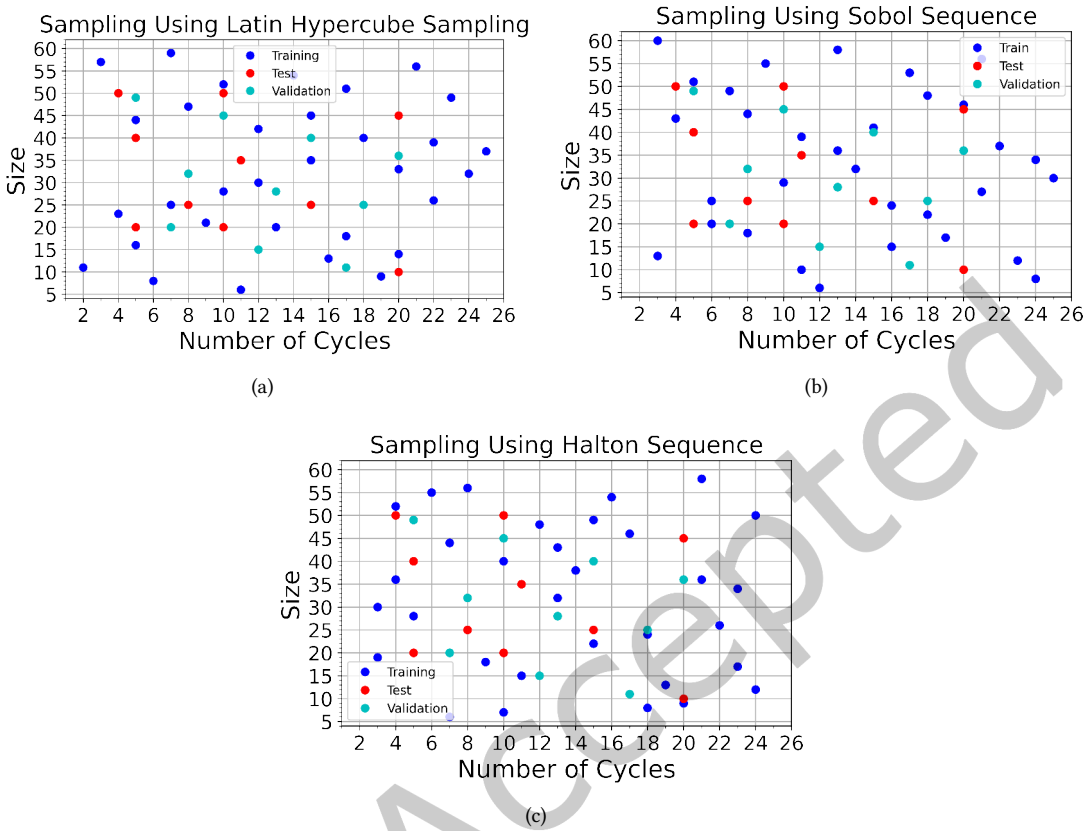


Fig. 10. Axiline architectural configurations (size and number of cycles) sampled using (a) Latin Hypercube Sampling, (b) Sobol Sequence, and (c) Halton Sequence. Blue, cyan, and red dots respectively denote training, validation, and testing configurations.

8 EXPERIMENTAL RESULTS

In this section, we present our experimental results. First, we show the performance of our model using various sampling methods and sample sizes. We then demonstrate the performance of the model for unseen backend and architectural configurations. Next, we study our model performance for limited training dataset. Finally, we utilize the trained model for the DSE of Axiline and VTA designs.

8.1 Assessment of Sampling Methods and Sample Sizes

The selection of an appropriate sampling method and sample size is crucial, as a poor choice may result in non-uniform sampling. This can subsequently lead to the development of biased models or necessitate a larger volume of data points in order to achieve the desired performance. To identify the appropriate sampling method and sample size, we train our models using data obtained from various sampling techniques and a range of sample sizes. Figure 10 presents the design configurations of Axiline generated using LHS, Sobol sequence and Halton sequence. Subsequently, we assess the performance of the trained models on unseen test configurations, capturing

the mean absolute percentage error (μAPE) and maximum absolute percentage error ($MAPE$). Additionally, we compute standard deviation of APE (STD APE) on the test configurations to measure stability of model performance on unseen configuration across the architectural parameter space. A smaller value of STD APE indicates stable model performance on the unseen configurations.

Table 3 shows the performance of different ML models for Axiline-SVM on the testing dataset when the training data is sampled using Latin Hypercube Sampling, Sobol Sequence, and Halton Sequence with sample sizes of 16, 24, and 32. For a given sample size and machine learning model, we use bold font to indicate the result of the top-performing sampling method. From Table 3, we make the following observations.

- Consistent with expectations, across all sampling methods, an increase in sample size leads to a decrease in both μAPE and STD APE, thus indicating an improvement in model performance.
- When considering smaller sample sizes, the GCN models demonstrate superior results in terms of both STD APE and $MAPE$ in comparison to other models.
- In 12/24 instances, LHS yields superior results in terms of μAPE, and in 10/24 instances, in terms of both $MAPE$ and STD APE, in comparison to other sampling techniques. Overall, LHS demonstrates better performance than either the Sobol or Halton methods. Consequently, we choose to use LHS in experiments reported below.
- The model's performance, across all metrics, does not exhibit substantial improvements when the sample size is increased from 24 to 32. Hence, for the experiments reported below, we maintain a sample size of 24.

To summarize: in the remaining portion of our experiments, we employ LHS with a sample size of 24 to generate the training dataset for Axiline designs.

8.2 ML Model Assessment

We now present the results of our model's performance for predicting backend PPA metrics, and system-level runtime and energy for both unseen backend and architectural configurations, over a set of benchmark workloads. Our performance evaluation is based on μAPE, $MAPE$ and STD APE. The latter helps us understand how consistently our model performs on the testing dataset.

Results for unseen backend configurations. Table 4 presents the performance of the ML model for unseen backend configurations in predicting post-SP&R PPA and system-level metrics for TABLA, Genesys, VTA, and Axiline designs, implemented on GF12 and/or NG45. For the ROI classification task, all models for the GF12 implementation achieve at least 96% accuracy and an F1 score of 0.97. For the Axiline NG45 implementation, all models achieve at least 94% accuracy and an F1 score of 0.96. These results demonstrate the excellent performance of our models in identifying whether data points belong to the ROI. In Table 4, the best-performing model based on μAPE for each design and each metric is highlighted in bold. Based on this table and for these models, we make the following observations.

- The best-performing ML model achieves a μAPE of less than 6.5% and a $MAPE$ of less than 30% for PPA prediction. For Axiline-NG45, the system-level energy prediction using the GCN model yields the highest $MAPE$. Upon investigating the model, we find that the STD APE on the testing dataset is 4.81.
- The best-performing ML model achieves a μAPE of less than 5% and a $MAPE$ of less than 38% for system-level metric prediction. For Axiline-NG45, the system-level runtime prediction using the ensemble model yields the highest $MAPE$. Upon investigating the model, we find that the STD APE on the testing dataset is 4.18.

As shown in Figure 7, we sample our test dataset so that it covers the design space uniformly. High accuracy and low μAPE on the test dataset suggests that our training dataset is large enough. Table 6 in Appendix A details the performance of GBDT, RF, and ANN models for unseen backend configurations.

Results for unseen architectural configurations. Table 5 shows the performance of the ML model for unseen architectural configurations in predicting post-SP&R PPA and system-level metrics for TABLA, Genesys, VTA, and Axiline designs, implemented on GF12 and/or NG45. For the ROI classification task all the models for GF12

Table 3. ML model performance on unseen architectural configurations for different sampling methods and sample sizes. Here, the standard deviation of *APE* (STD *APE*) represents the variation in *APE* across all different test configurations. A smaller value of STD *APE* indicates stable performance on unseen architectural configurations.

Sampling Details		ML Model	Power			System-Energy		
Method	Size		μAPE	STD *APE*	*MAPE*	μAPE	STD *APE*	*MAPE*
LHS	16	GBDT	20.37	**13.14**	84.16	32.65	25.68	88.69
		RF	**17.63**	**10.01**	**57.00**	**36.58**	23.28	**78.84**
		ANN	**2.67**	**1.31**	23.02	**4.36**	2.95	28.13
		GCN	**2.96**	**0.45**	**15.89**	**3.06**	0.99	**13.92**
	24	GBDT	13.20	8.07	**58.63**	**15.25**	**9.92**	**65.02**
		RF	14.76	12.29	84.38	**14.38**	12.07	68.17
		ANN	**1.80**	**0.54**	15.83	3.44	**2.26**	**21.73**
		GCN	3.00	0.91	**15.38**	2.71	0.74	16.21
	32	GBDT	13.61	6.07	59.70	**9.75**	**6.69**	48.34
		RF	12.06	**6.73**	44.10	21.88	20.00	84.68
		ANN	**2.03**	0.70	**13.24**	4.00	3.65	31.51
		GCN	**2.57**	0.70	**15.99**	2.20	**0.66**	19.92
Sobol	16	GBDT	**18.02**	15.64	**72.16**	**28.19**	**16.12**	**82.50**
		RF	22.58	15.07	75.50	39.63	**15.15**	85.97
		ANN	3.18	1.52	**20.67**	5.16	**2.15**	**24.11**
		GCN	3.24	0.94	18.03	3.32	**0.87**	22.39
	24	GBDT	14.31	10.65	80.92	34.14	33.93	99.20
		RF	18.32	13.78	73.92	29.41	19.15	89.63
		ANN	2.70	1.31	27.40	5.19	2.45	22.21
		GCN	**2.51**	1.05	15.89	2.62	**0.69**	**15.85**
	32	GBDT	14.95	9.98	37.90	21.45	16.52	**34.89**
		RF	16.06	12.74	**33.85**	25.84	27.19	**46.02**
		ANN	2.39	1.00	25.07	2.59	1.46	21.31
		GCN	2.58	0.72	18.03	**2.14**	0.72	18.87
Halton	16	GBDT	19.28	15.32	80.90	48.54	43.57	85.08
		RF	21.46	18.24	88.02	49.52	79.62	85.01
		ANN	4.07	2.57	37.25	9.22	8.23	60.09
		GCN	3.81	1.46	18.38	4.31	1.30	18.38
	24	GBDT	**12.80**	**7.80**	69.04	26.27	16.46	79.35
		RF	**13.15**	**10.63**	**49.89**	20.57	**12.01**	**61.67**
		ANN	1.94	0.57	**13.31**	**3.01**	2.50	32.13
		GCN	2.65	**0.40**	17.07	**2.51**	0.71	16.38
	32	GBDT	**8.88**	**3.80**	**32.64**	27.48	24.46	95.76
		RF	**11.46**	7.65	40.66	**21.48**	**13.39**	58.25
		ANN	2.27	**0.59**	21.03	**2.44**	**1.08**	**18.07**
		GCN	2.74	**0.58**	19.73	2.71	0.84	**15.98**

implementation achieve at least 95% accuracy and 0.97 F1 score. This indicates that models are performing very well in determining whether the data points belong to the ROI. In Table 5, the best-performing model based on μAPE for each design and each metric is highlighted in bold. Based on this table and for these models, we make the following observations.

Table 4. Performance of ML models for unseen backend configurations.

Design	ML Model	Performance		Power		Area		System-Energy		System-Runtime	
		μAPE	*MAPE*	*μAPE*	*MAPE*	*μAPE*	*MAPE*	*μAPE*	*MAPE*	*μAPE*	*MAPE*
TABLA	Ensemble	2.82	11.00	2.28	9.51	1.25	6.33	**0.93**	3.53	3.84	14.55
GF12	GCN	**2.75**	11.56	**2.18**	8.94	**0.54**	5.64	0.93	5.39	**3.03**	11.82
GeneSys	Ensemble	8.38	24.36	**6.45**	22.02	1.00	3.04	**1.80**	5.37	6.45	17.86
GF12	GCN	**6.00**	20.59	7.28	15.81	**0.49**	1.30	**1.80**	5.11	**5.83**	15.51
VTA	Ensemble	2.79	14.57	2.67	11.94	1.15	4.35	**2.36**	10.43	4.07	12.04
GF12	GCN	**2.16**	12.21	**2.18**	7.77	**0.66**	4.02	2.46	6.92	**2.31**	8.53
Axiline	Ensemble	**0.70**	8.50	2.44	28.53	**1.46**	20.99	9.15	95.32	**1.05**	8.30
GF12	GCN	3.06	49.65	**1.52**	22.69	1.82	16.09	**2.68**	37.83	1.39	25.56
Axiline	Ensemble	**3.15**	23.61	7.68	54.21	**1.39**	8.81	8.91	75.83	**5.16**	31.31
NG45	GCN	4.74	36.25	**5.19**	29.98	3.03	13.48	**4.97**	25.07	**4.59**	55.06

- The best-performing ML model achieves a *μAPE* of less than 7% and a *MAPE* of less than 37% for backend PPA prediction. For Axiline-NG45, the power prediction using the GCN model yields the highest *MAPE*. Upon examining the model, we find that the STD *APE* on the testing dataset is 4.49.
- The best-performing ML model achieves a *μAPE* of less than 8% and a *MAPE* of less than 50% for system-level metric prediction. For Axiline-NG45, the system-level energy prediction using the Ensemble model yields the highest *MAPE*. Upon investigating the model, we find that the STD *APE* on the testing dataset is 7.07.

Table 5. Performance of ML models for unseen architectural configurations.

Design	ML Model	Performance		Power		Area		System-Energy		System-Runtime	
		μAPE	*MAPE*	*μAPE*	*MAPE*	*μAPE*	*MAPE*	*μAPE*	*MAPE*	*μAPE*	*MAPE*
TABLA	Ensemble	**3.68**	32.51	**4.11**	17.11	3.99	16.05	4.62	18.63	6.03	24.10
GF12	GCN	5.79	21.74	5.34	14.00	**3.76**	12.81	**3.93**	13.80	**5.20**	23.63
GeneSys	Ensemble	**6.32**	14.82	7.26	15.23	2.75	8.09	11.96	19.78	**6.28**	20.27
GF12	GCN	6.97	13.11	**5.39**	15.10	**2.12**	4.08	**4.32**	8.88	7.65	17.81
VTA	Ensemble	2.99	12.58	11.19	28.99	6.65	17.01	9.10	18.41	**2.87**	10.50
GF12	GCN	**2.60**	9.67	**2.85**	12.90	**2.15**	9.51	**4.07**	13.76	3.67	9.87
Axiline	Ensemble	**0.61**	6.48	**2.55**	22.45	**1.31**	5.68	7.17	47.20	**1.29**	7.74
GF12	GCN	2.92	28.74	2.86	29.34	1.88	9.82	**2.34**	29.21	2.98	2971
Axiline	Ensemble	**3.33**	27.29	5.97	48.34	**2.81**	17.25	**7.21**	49.81	**4.75**	22.63
NG45	GCN	4.57	35.68	**5.55**	36.38	3.77	16.26	12.88	86.2	5.85	51.21

For some unseen architectural configurations, we observe MAPE values higher than 30%. However, further investigation reveals smaller standard deviation values for the *APE*. This indicates that our model delivers reliable results for most of the data points. Table 7 in Appendix A details the performance of GBDT, RF, and ANN models for unseen architectural configurations. We also observe that for both unseen backend and unseen architectural configuration datasets, the GCN model outperforms other models in most scenarios. In instances where it is not the best model, it still yields results very similar to those of the best-performing model.

8.3 Effect of Limited Training Dataset

We have additionally studied the performance of our model in terms of *extrapolation*. To put it more precisely, we study how well our model performs when the test data points fall outside the range of the training dataset. For

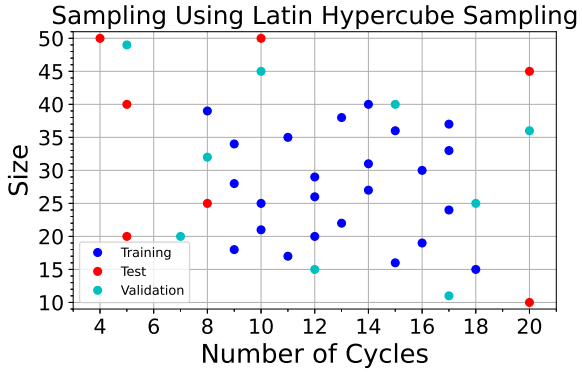


Fig. 11. Sampled train, validation and test data points for extrapolation experiment. Here, size and number of cycles are the architectural parameters for Axiline design.

this experiment, we sample architectural configurations of the Axiline design as displayed in Figure 11, where the blue, red, and cyan data points correspond to training, testing, and validation data respectively. For each architectural configuration, we run 30 SP&R jobs to prepare the training, testing, and validation datasets. We observe that the model performs poorly on the validation dataset and produces similar poor results for the testing dataset, confirming the limitations of the ML model we use for dataset extrapolation.

However, our training dataset covers the architectural design space for Axiline and backend design space for all accelerators including GeneSys, VTA, TABLA and Axiline. The count of features handled by the Axiline design is computed as $num\_cycles \times size$. The Axiline designs are expected to handle up to 800 features [41] and our sample space already covers up to 900 features. As seen in Figure 4, the $f_{effective}$ does not change for higher f_{target}, indicating that we have effectively covered the backend design space. Therefore, even though the models fail to produce satisfactory results for the extrapolation dataset, it is not considered a drawback. Our training's configuration space includes the entire design space, eliminating the need for the model to function outside the scope of the training configurations.

8.4 DSE with Trained ML Models

We now apply our trained models for DSE on two different platforms: Axiline and VTA. We apply the MOTPE method and our trained models for the DSE of the ML accelerator, with the objective of minimizing chip area and system energy. We also employ an additional flag during the DSE process to indicate whether the explored data points fulfill design configuration, runtime, and power requirements. The specifics of these two DSE experiments are as follows.

DSE of Axiline-SVM in NG45 enablement. We optimize the implementation of an accelerator that executes the SVM algorithm with 55 features. The flow is as follows.

- We select a range for architectural and backend parameter configurations.
- For the given set of configurations, we carry out DSE using MOTPE and trained models, capturing energy, runtime, total power, and chip area for the sampled configurations.
- We identify the best configuration that minimizes Equation (3) when α is 0.001.
- For the best configuration, we generate the RTL netlist, run SP&R and collect the actual backend and chip parameters.

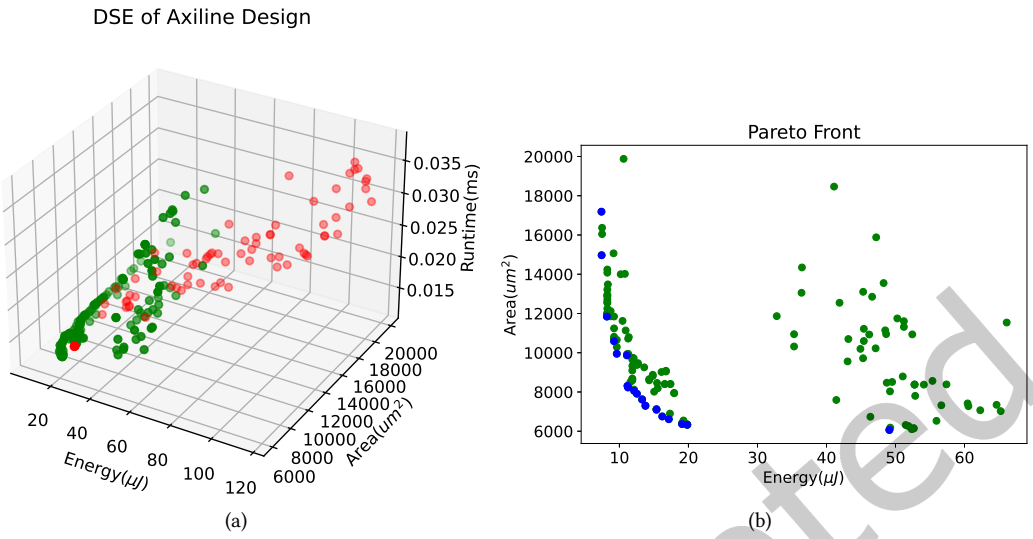


Fig. 12. Design space exploration of the Axiline-SVM designs. (a) Energy, runtime, and area metrics of the explored data points. Data points highlighted in red fail to meet the ROI, power, and runtime criteria. (b) Area vs. energy plot for the explored data points. Data points highlighted in blue belong to the Pareto front.

During the DSE process, we vary *size* from 10 to 51, *num_cycle* from 5 to 21, f_{target} from 0.3 to 1.3, and floorplan utilization from 0.4 to 0.8. Figure 12(a) displays the predicted runtime, area, and energy plot for all the data points sampled during the DSE process. The data points highlighted with red dots do not meet the ROI, power and runtime requirements. We then identify the best configuration that minimizes Equation (3). Figure 12(b) presents Pareto front of area versus energy plot. For ground truth analysis, we also generate the RTL netlist and run SP&R for each of the top three configurations, and confirm that the predicted metrics for these top three configurations are within 7% of post-SP&R values.

DSE of VTA design in GF12 enablement. We also apply our the DSE method to optimize the backend configuration for the VTA design. The process is the same as outlined above for Axiline-SVM, except we only vary f_{target} from 0.3 to 1.3 and floorplan utilization from 0.25 to 0.55. Figure 13(a) displays the predicted runtime, area, and energy plots for all the data points sampled during the DSE process. The data points marked with red dots do not meet the ROI, power, and runtime requirements. We identify the best configuration that minimizes Equation (3) when α is 1. We adjust these values because the units of energy and runtime in Figure 13(a) differ from those in Figure 12(a). Figure 13(b) presents Pareto front of area versus energy plot. Running SP&R on the top three backend configurations identified from the DSE confirms that the predicted metrics for these top three configurations are within 6% of the post-SP&R values.

9 CONCLUSION

We introduce a physical-design-driven, ML-based framework that consistently predicts backend PPA and system metrics with an average 7% or less prediction error for the ASIC implementation of two deep learning accelerator platforms: VTA and VeriGOOD-ML, in both a commercial 12 nm process and a research-oriented 45 nm process. The framework integrates several ML modeling aspects, including a focus on the “region of interest", a novel

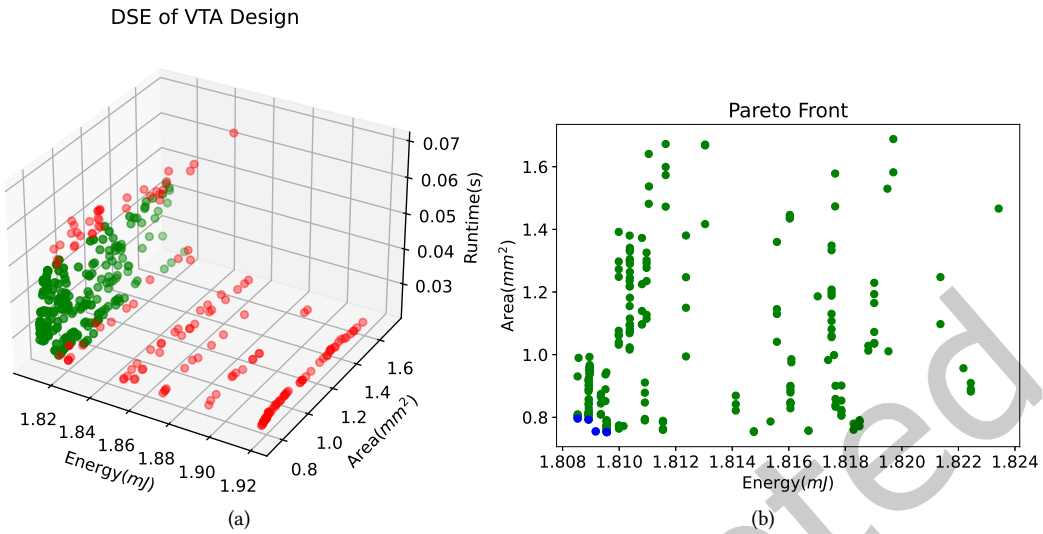


Fig. 13. Backend design space exploration of a VTA design. (a) Energy, runtime, and area metrics of the explored data points. Data points highlighted in red fail to meet the ROI, power, and runtime criteria. (b) Area vs. energy plot for the explored data points. Data points highlighted in blue belong to the Pareto front.

two-stage approach, and the employment of logical hierarchy graphs for a GCN model, enabling efficient model-guided MOTPE-based automated searches over vast accelerator architecture and backend configuration spaces for a given workload or ML algorithm. We extensively validate our framework on multiple ML accelerator platforms.

ACKNOWLEDGMENTS

This material is based on research sponsored in part by Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) under agreement number FA8650-20-2-7009. Andrew B. Kahng also acknowledges support from NSF CCF-2112665. The U. S. government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of AFRL, DARPA, or the U. S. government.

The authors would like to acknowledge the contributions of Steven M. Burns and Anton A. Sorokin from Intel Labs.

REFERENCES

[1] A. Agnesina, K. Chang and S. K. Lim, "VLSI Placement Parameter Optimization using Deep Reinforcement Learning", *Proc. ICCAD*, 2020, pp. 1–9.

[2] C. Bai, Q. Sun, J. Zhai, Y. Ma, B. Yu and M. D. F. Wong, "BOOM-Explorer: RISC-V BOOM Microarchitecture Design Space Exploration Framework", *Proc. ICCAD*, 2021.

[3] S. Banerjee, S. Burns, P. Cocchini, A. Davare, S. Jain, D. Kirkpatrick et al., "A Highly Configurable Hardware/Software Stack for DNN Inference Acceleration", *arXiv:2111.15024*, 2020.

[4] J. Bergstra and Y. Bengio, "Random Search for Hyper-parameter Optimization", *Journal of Machine Learning Research*, 13(10), 2012, pp. 281–305.

[5] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, M. Cowan et al., "TVM: An Automated End-to-End Optimizing Compiler for Deep Learning", *Proc. OSDI*, 2018, pp. 578—594.

[6] Y. -H. Chen, T. Krishna, J. S. Emer and V. Sze, "Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks," *IEEE JSSC*, 52(1) (2017), pp. 127-138.
[7] C. K. Cheng, C. Holtz, A. B. Kahng, B. Lin and U. Mallappa, "DAGSizer: A Directed Graph Convolutional Network Approach to Discrete Gate Sizing of VLSI Graphs", *ACM TODAES*, 28(4), 2023 pp. 1–31.
[8] S. Dai, Y. Zhou, H. Zhang, E. Ustun, E. F. Y. Young and Z. Zhang, "Fast and Accurate Estimation of Quality of Results in High-Level Synthesis with Machine Learning", *Proc. FCCM*, 2018, pp. 129–132.
[9] H. Esmaeilzadeh, S. Ghodrati, J. Gu, S. Guo, A. B. Kahng, J. K. Kim et al., "VeriGOOD-ML: An Open-Source Flow for Automated ML Hardware Synthesis", *Proc. ICCAD*, 2021, pp. 1–8.
[10] H. Esmaeilzadeh, S. Ghodrati, A. B. Kahng, J. K. Kim, S. Kinzer, S. Kundu et al., "Physically Accurate Learning-based Performance Prediction of Hardware-accelerated ML Algorithms", *Proc. MLCAD*, 2022.
[11] M. Fey and J. E. Lenssen, "Fast Graph Representation Learning with PyTorch Geometric", *Proc. ICLR*, 2019.
[12] H. Genc, S. Kim, A. Amid, A. H.-Ali, V. Iyer, P. Prakash et al., "Gemmini: Enabling Systematic Deep-Learning Architecture Evaluation via Full-Stack Integration", *Proc. DAC*, 2021, pp. 769–774.
[13] T. Head, MechCoder, G. Louppe, I. Shcherbatyi, A. Fabisch et al. scikit-optimize/scikit-optimize: v0.5.2, *Zenodo*, 2018, http://doi.org/10.5281/zenodo.1207017.
[14] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa et al., "In-datacenter Performance Analysis of a Tensor Processing Unit", *Proc. ISCA*, 2017, pp. 1—12.
[15] A. B. Kahng, B. Lin and S. Nath, "ORION3.0: A Comprehensive NoC Router Estimation Tool", *IEEE Embedded Systems Letters* 7(2) (2015), pp. 41–45.
[16] D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization", *Proc. ICLR*, 2014.
[17] M. J. van der Laan, E. C Polley and A. E. Hubbard, "Super Learner", Statistical Applications in Genetics and Molecular Biology. 2007; 6(1). https://doi.org/10.2202/1544-6115.1309
[18] F. Last and U. Schlichtmann, "Feeding Hungry Models Less: Deep Transfer Learning for Embedded Memory PPA Models", *Proc. MLCAD*, 2021, pp. 1–6.
[19] W. Lee, Y. Kim, J. H. Ryoo, D. Sunwoo, A. Gerstlauer and L. K. John, "PowerTrain: A Learning-based Calibration of McPAT Power Models", *Proc. ISLPED*, 2015, pp. 189–194.
[20] F. Li, Y. Wang, C. Liu, H. Li and X. Li, "NoCeption: A Fast PPA Prediction Framework for Network-on-Chips using Graph Neural Network", *Proc. DATE*, 2022.
[21] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen and N. P. Jouppi, "McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures", *Proc. MICRO*, 2009.
[22] S. D. Manasi, F. S. Snigdha and S. S. Sapatnekar, "NeuPart: Using Analytical Models to Drive Energy-Efficient Partitioning of CNN Computations on Cloud-Connected Mobile Clients," *IEEE TVLSI*, 28(8) (2020), pp. 1844-1857.
[23] R. Liaw, E. Liang, R. Nishihara, P. Moritz, J. E. Gonzalez and I. Stoica, "Tune: A Research Platform for Distributed Model Selection and Training", *arXiv preprint*, 2018.
[24] Z. Lin, J. Zhao, S. Sinha and W. Zhang, "HL-Pow: A Learning-Based Power Modeling Framework for High-Level Synthesis", *Proc. ASP-DAC*, 2020, pp. 574–580.
[25] Y.-C. Lu, W.-T. Chan, V. Khandelwal and S. K. Lim, "Driving Early Physical Synthesis Exploration through End-of-Flow Total Power Prediction", *Proc. MLCAD*, 2022.
[26] D. Mahajan, J. Park, E. Amaro, H. Sharma, A. Yazdanbakhsh, J. K. Kim et al., "TABLA: A Unified Template-based Framework for Accelerating Statistical Machine Learning", *Proc. HPCA*, 2016, pp. 14–26.
[27] S. D. Manasi, F. S. Snigdha and S. S. Sapatnekar, "NeuPart: Using Analytical Models to Drive Energy-Efficient Partitioning of CNN Computations on Cloud-Connected Mobile Clients", *IEEE TVLSI* 28(8) (2018), pp. 1844–1857.
[28] T. Moreau, T. Chen, L. Vega, J. Roesch, E. Yan, L. Zheng et al., "A Hardware–Software Blueprint for Flexible Deep Learning Specialization", *IEEE Micro*, 39(5) (2019), pp. 8–16.
[29] S. D. Manasi and S. S. Sapatnekar, "DeepOpt: Optimized Scheduling of CNN Workloads for ASIC-based Systolic Deep Learning Accelerators", in *Proc. ASPDAC*, 2021, pp. 235–241.
[30] H. Niederreiter, "Low-Discrepancy and Low-Dispersion Sequences", *Journal of number theory*, 30(1), 1988, pp. 51–70.
[31] Y. Ozaki, Y. Tanigaki, S. Watanabe and M. Onishi, "Multiobjective Tree-Structured Parzen Estimator for Computationally Expensive Optimization Problems", *Proc. GECCO*, 2020.
[32] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan et al., "PyTorch: An Imperative Style, High-Performance Deep Learning Library", *Proc. NeurIPS*, 2019.
[33] P. Sengupta, A. Tyagi, Y. Chen and J. Hu, "How Good Is Your Verilog RTL Code? A Quick Answer from Machine Learning", *Proc. ICCAD*, 2022.
[34] Y. S. Shao, B. Reagen, G. -Y. Wei and D. Brooks, "Aladdin: A Pre-RTL, Power-Performance Accelerator Simulator Enabling Large Design Space Exploration of Customized Architectures", *Proc. ISCA*, 2014, pp. 97–108.

[35] E. Tabanelli, G. Tagliavini and L. Benini, "DNN Is Not All You Need: Parallelizing Non-Neural ML Algorithms on Ultra-Low-Power IoT Processors", *arXiv:2107.09448*, 2021. https://arxiv.org/abs/2107.09448.
[36] S. Takamaeda-Yamazaki, "Pyverilog: A Python-Based Hardware Design Processing Toolkit for Verilog HDL", *Applied Reconfigurable Computing*, 2015.
[37] H.-S. Wang, X. Zhu, L.-S. Peh and S. Malik, "Orion: A Power-Performance Simulator for Interconnection Networks", *Proc. MICRO*, 2002, pp. 294–395.
[38] S. Williams and M. Baxter, "Icarus Verilog: Open-source Verilog More Than a Year Later", *Linux Journal*, 2002.
[39] P. Xu, X. Zhang, C. Hao, Y. Zhao, Y. Zhang, Y. Wang et al., "AutoDNNchip: An Automated DNN Chip Predictor and Builder for Both FPGAs and ASICs", *Proc. FPGA*, 2020, pp. 40–50.
[40] Z. Zeng and S. S. Sapatnekar, "Energy-efficient Hardware Acceleration of Shallow Machine Learning Applications", *Proc. DATE*, 2023.
[41] VeriGood-ML, https://github.com/VeriGOOD-ML/public.
[42] AutoML: Automatic Machine Learning, https://docs.h2o.ai/h2o/latest-stable/h2o-docs/automl.html.
[43] "VTA Hardware Design Stack", https://github.com/pasqoc/incubator-tvm-vta.
[44] GitHub repository: "VeriGOOD-ML: Verilog Generator, Optimized for Designs for Machine Learning". https://github.com/VeriGOOD-ML/public
[45] NanGate45 PDK. https://eda.ncsu.edu/freepdk/freepdk45/

A ML MODEL ASSESSMENT

In this appendix, we present the performance of GBDT, RF, and ANN models, along with the Ensemble model and GCN model, for predicting PPA, as well as system-level energy and runtime for TABLA, GeneSys, VTA, and Axiline designs on GF12 or Nangate45 enablement. Table 6 details the model performance for unseen backend configurations, while Table 7 showcases the model performance for unseen architectural configurations.

Received 22 August 2023; revised 4 January 2024; accepted 13 March 2024

Table 6. Performance of ML models for unseen backend configurations.

Design	ML Model	Performance		Power		Area		System-Energy		System-Runtime	
		μAPE	*MAPE*	*μAPE*	*MAPE*	*μAPE*	*MAPE*	*μAPE*	*MAPE*	*μAPE*	*MAPE*
TABLA GF12	GBDT	3.09	14.02	2.88	11.94	0.78	3.02	1.22	5.15	3.44	13.28
	RF	6.13	29.43	3.58	12.15	2.59	10.55	1.83	5.08	6.17	20.39
	ANN	3.21	11.05	2.88	13.36	**0.24**	0.93	**0.85**	2.73	3.14	17.15
	Ensemble	2.82	11.00	2.28	9.51	1.25	6.33	0.93	3.53	3.84	14.55
	GCN	**2.75**	11.56	**2.18**	8.94	0.54	5.64	0.93	5.39	**3.03**	11.82
GeneSys GF12	GBDT	7.16	22.74	8.57	26.41	3.93	12.82	1.50	7.03	6.76	20.15
	RF	11.04	23.52	8.48	18.29	3.25	7.46	1.56	7.60	8.99	34.54
	ANN	6.50	15.49	**5.26**	17.93	0.84	2.27	2.80	7.80	6.40	18.46
	Ensemble	8.38	24.36	6.45	22.02	1.00	3.04	**1.80**	5.37	6.45	17.86
	GCN	**6.00**	20.59	7.28	15.81	**0.49**	1.30	**1.80**	5.11	**5.83**	15.51
VTA GF12	GBDT	2.75	13.38	2.84	12.75	1.90	13.14	**1.89**	8.27	2.84	12.18
	RF	5.67	35.31	4.57	28.02	2.66	12.64	2.07	7.58	4.68	24.74
	ANN	2.29	14.00	**2.05**	8.59	0.89	3.85	7.29	24.37	2.47	10.54
	Ensemble	2.79	14.57	2.67	11.94	1.15	4.35	2.36	10.43	4.07	12.04
	GCN	**2.16**	12.21	2.18	7.77	**0.66**	4.02	2.46	6.92	**2.31**	8.53
Axiline GF12	GBDT	0.77	5.24	2.20	14.70	2.74	13.59	1.34	12.31	1.15	8.87
	RF	6.55	36.06	3.79	29.70	3.50	16.94	**1.32**	13.00	7.53	91.34
	ANN	0.78	8.69	2.78	28.19	2.21	53.32	4.46	77.34	1.29	13.16
	Ensemble	**0.70**	8.50	2.44	28.53	**1.46**	20.99	9.15	95.32	**1.05**	8.30
	GCN	3.06	49.65	**1.52**	22.69	1.82	16.09	2.68	37.83	1.39	25.56
Axiline NG45	GBDT	3.56	22.73	7.01	33.61	2.60	12.62	6.43	51.06	**3.95**	36.78
	RF	4.56	30.57	9.38	45.35	3.70	13.38	6.92	41.56	4.21	44.36
	ANN	3.48	25.40	8.48	83.22	1.93	25.85	7.04	51.25	6.60	45.50
	Ensemble	**3.15**	23.61	7.68	54.21	**1.39**	8.81	8.91	75.83	5.16	31.31
	GCN	4.74	36.25	**5.19**	29.98	3.03	13.48	**4.97**	25.07	4.59	55.06

Table 7. Performance of ML models for unseen architectural configurations.

Design	ML Model	Performance		Power		Area		System-Energy		System-Runtime	
		μAPE	*MAPE*	*μAPE*	*MAPE*	*μAPE*	*MAPE*	*μAPE*	*MAPE*	*μAPE*	*MAPE*
TABLA GF12	GBDT	**3.24**	33.06	**3.87**	19.03	3.42	9.01	8.86	19.15	**3.83**	18.99
	RF	5.16	38.01	9.76	46.62	11.25	40.13	10.59	21.30	5.67	27.41
	ANN	5.78	32.70	5.22	20.02	**2.30**	5.70	**2.97**	10.25	6.02	24.94
	Ensemble	3.68	32.51	4.11	17.11	3.99	16.05	4.62	18.63	6.03	24.10
	GCN	5.79	21.74	5.34	14.00	3.76	12.81	3.93	13.80	5.20	23.63
GeneSys GF12	GBDT	6.54	20.86	5.82	15.99	2.94	8.52	6.37	11.65	12.23	28.41
	RF	8.73	22.57	9.34	18.66	3.69	9.57	14.89	22.00	17.37	34.38
	ANN	6.55	19.86	**3.82**	15.36	**2.06**	3.80	**3.47**	9.11	10.73	28.90
	Ensemble	**6.32**	14.82	7.26	15.23	2.75	8.09	11.96	19.78	**6.28**	20.27
	GCN	6.97	13.11	5.39	15.10	2.12	4.08	4.32	8.88	7.65	17.81
VTA GF12	GBDT	4.96	19.31	4.04	11.83	24.74	38.07	7.58	18.79	6.94	20.39
	RF	3.00	9.81	12.96	33.34	18.05	53.58	7.15	16.43	5.67	13.52
	ANN	**2.52**	14.09	3.08	11.84	2.19	6.66	10.61	22.16	4.39	12.70
	Ensemble	2.99	12.58	11.19	28.99	6.65	17.01	9.10	18.41	**2.87**	10.50
	GCN	2.60	9.67	**2.85**	12.90	**2.15**	9.51	**4.07**	13.76	3.67	9.87
Axiline GF12	GBDT	0.62	7.18	11.53	74.19	10.29	41.78	16.61	82.95	2.19	19.83
	RF	0.63	5.41	15.95	77.38	13.24	57.18	21.8	90.34	2.12	12.86
	ANN	0.72	8.64	**2.24**	21.98	**1.20**	7.85	4.24	29.85	**1.08**	9.72
	Ensemble	**0.61**	6.48	2.55	22.45	1.31	5.68	7.17	47.20	1.29	7.74
	GCN	2.92	28.74	2.86	29.34	1.88	9.82	**2.34**	29.21	2.98	2971
Axiline NG45	GBDT	3.45	27.48	5.98	56.62	2.79	13.40	23.33	90.79	5.54	30.13
	RF	**3.18**	26.96	6.30	41.31	3.00	16.90	21.54	93.74	5.77	35.33
	ANN	3.37	27.19	6.57	59.76	**1.86**	13.81	9.30	77.10	5.04	25.56
	Ensemble	3.33	27.29	5.97	48.34	2.81	17.25	**7.21**	49.81	**4.75**	22.63
	GCN	4.57	35.68	**5.55**	36.38	3.77	16.26	12.88	86.2	5.85	51.21