University of California
Santa Barbara

# Machine Learning Empowered Agile Hardware Design and Design Automation

A dissertation submitted in partial satisfaction
of the requirements for the degree

Doctor of Philosophy
in
Electrical and Computer Engineering

by

Nan Wu

Committee in charge:

Professor Yuan Xie, Co-Chair
Professor Peng Li, Co-Chair
Professor Li-C Wang
Professor Timothy Sherwood
Professor Zheng Zhang
Professor Yufei Ding
Professor Cong Hao

June 2023

The Dissertation of Nan Wu is approved.

---

Professor Li-C Wang

---

Professor Timothy Sherwood

---

Professor Zheng Zhang

---

Professor Yufei Ding

---

Professor Cong Hao

---

Professor Peng Li, Committee Co-Chair

---

Professor Yuan Xie, Committee Co-Chair

May 2023

Machine Learning Empowered Agile Hardware Design and Design Automation



by

Nan Wu

# Acknowledgements

**To all the amazing people sharing experiences and thoughts with me**

**Thank you for shaping my philosophy on research and life**

Over the past few years, I am so privileged to have met many exceptional teachers and mentors, who guide me right paths to career and life, encourage me to pursue my goals, and vigorously support me all the way. I will not be where I am today without standing on their shoulders.

I am incredibly grateful to my advisor, Prof. Yuan Xie, for many years of guidance and support throughout my academic journey, for always being available and present, for being so professional and caring at the same time, and for being a constant source of inspiration. I have learned a lot from your broad knowledge and big vision in research, as well as your wisdom and passion for life. You are certainly a role model for me in how to cultivate a successful career and how to become an exceptional person. You have taught me so much more than research skills and I aspire to become what you are to me to the next-generation blooming researchers. There is a Chinese proverb that states, "A teacher for a day is to be respected as a father for a lifetime". I am blessed with the best advisor from whom I can learn valuable lessons lifelong.

I would like to convey my heartfelt gratitude to my thesis committee members, Prof. Peng Li, Prof. Li-C Wang, Prof. Timothy Sherwood, Prof. Zheng Zhang, Prof. Yufei Ding, and Prof. Cong Hao. Thank you, Prof. Peng Li, for being so patient, careful, and helpful throughout the entire journey. Thank you, Prof. Li-C Wang, for enlightening me with new research perspectives to examine the question that I have not thought about. Thank you, Prof. Timothy Sherwood, for your unwavering support and providing me with valuable insights for my research. Thank you, Prof. Zheng Zhang, for showing me the allure of conducting rigorous and meticulous research. Thank you, Prof. Yufei Ding,

for your helpfulness and providing me with invaluable feedback on my research. Thank you, Prof. Cong Hao, for being receptive to my random ideas, encouraging me to keep being proactive, and inspiring me to live bright and courageous womanhood.

I am deeply indebted to all of my collaborators, Prof. Cong Hao, Prof. Cunxi Yu, Prof. Pan Li, Prof. Guoqi Li, and Prof. Lei Deng for their knowledgeable advice and insightful guidance. I sincerely thank the coauthors of my research papers, including but not limited to Prof. Dmitri Strukov, Dr. Steve Dai, Dr. Adrien Vincent, Yingjie Li, Haoyu Wang, Hang Yang, and Hanqiu Chen.

I have been so fortunate to meet so many wonderful, intelligent, and diligent researchers at UCSB Seal-Lab, including Dr. Ping Chi, Dr. Shuangchen Li, Dr. Xing Hu, Dr. Lei Deng, Dr. Fengbin Tu, Dr. Jiayi Huang, Dr. Maohua Zhu, Dr. Peng Gu, Dr. Dylan Stow Randall, Dr. Itir Akgun, Dr. Wenqin Huangfu, Dr. Ling Liang, Dr. Xinfeng Xie, Dr. Tianqi Tang, Dr. Gushu Li, Bangyan Wang, Dr. Zheng Qu, Dr. Jilan Lin, Zhaodong Chen, Guyue Huang, Hao Li, Siqi Li, and Zhaohui Yang. Many thanks to my previous roommates, Dr. Beihang Yu, Dr. Yuning Shen, Dr. Jiren Zeng, and Dr. Ganghua Mei, for those fun memories.

Last but not the least, I would like to express my deepest gratitude to my parents. Thank you, Mom and Dad, for continuously encouraging me and cheering me on in my academic and personal pursuits. I cannot thank you enough for always being there to lift me up when I am down and believing in me when I lose faith in myself. I owe who I am today to your unconditional love and unwavering support.

**"No one else could ever be admitted here,**

**since this gate was made only for you."**

— Franz Kafka

May I continue to strive to live up to my dreams in the years to come.

# Curriculum Vitæ

Nan Wu

## Education

| | |
|---|---|
| 2023 | Ph.D. in Electrical and Computer Engineering (Expected), University of California, Santa Barbara. |
| 2018 | M.S. in Electrical and Computer Engineering, University of California, Santa Barbara. |
| 2016 | B.S. in Electronic Engineering, Tsinghua University. |

## Publications

[1] **Nan Wu**, Yingjie Li, Cong Hao, Steve Dai, Cunxi Yu, Yuan Xie, "GAMORA: Graph Learning based Symbolic Reasoning for Large-Scale Boolean Networks", Proc. *the 60th ACM/IEEE Design Automation Conference (DAC)*, 2023.

[2] Lakshmi Sathidevi, Abhinav Sharma, **Nan Wu**, Xun Jiao, Cong Hao, "PreAxC: Error Distribution Prediction for Approximate Computing Quality Control using Graph Neural Networks", Proc. *International Symposium on Quality Electronic Design (ISQED)*, 2023.

[3] **Nan Wu**, Yuan Xie, Cong Hao, "AI-assisted Synthesis in Next Generation EDA: Promises, Challenges, and Prospects", Proc. *IEEE International Conference on Computer Design (ICCD)*, 2022. (Invited)

[4] Haoyu Wang, **Nan Wu**, Hang Yang, Cong Hao, Pan Li, "Unsupervised Learning for Combinatorial Optimization with Principled Objective Relaxation", Proc. *the 36th Annual Conference on Neural Information Processing Systems (NeurIPS)*, 2022.

[5] **Nan Wu**, Hang Yang, Yuan Xie, Pan Li, Cong Hao, "High-Level Synthesis Performance Prediction using GNNs: Benchmarking, Modeling, and Advancing", Proc. *the 59th ACM/IEEE Design Automation Conference (DAC)*, 2022.

[6] **Nan Wu**, Yuan Xie, Cong Hao, "IronMan-Pro: Multi-objective Design Space Exploration in HLS via Reinforcement Learning and Graph Neural Network based Modeling", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2022.

[7] **Nan Wu**, Jiwon Lee, Yuan Xie, Cong Hao, "LOSTIN: Logic Optimization via Spatio-Temporal Information with Hybrid Graph Models", Proc. *the 33rd IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2022. **Best Paper Candidate**

[8] **Nan Wu**, Yuan Xie, "A Survey of Machine Learning for Computer Architecture and Systems", *ACM Computing Surveys (CSUR)*, 2022.

[9] **Nan Wu**, Yuan Xie, Cong Hao, "IronMan: GNN-assisted Design Space Exploration in High-Level Synthesis via Reinforcement Learning", Proc. *Great Lakes Symposium on VLSI (GLSVLSI)*, 2021. **Best Paper Award**

[10] **Nan Wu**, Lei Deng, Guoqi Li, Yuan Xie, "Core Placement Optimization for Multi-chip Many-core Neural Network Systems with Reinforcement Learning", *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 2020.

[11] **Nan Wu**, Adrien Vincent, Dmitri Strukov, "Preliminary Results Towards Reinforcement Learning with Mixed-Signal Memristive Neuromorphic Circuits", Proc. *IEEE International Symposium on Circuits and Systems (ISCAS)*, 2019.

[12] Behrooz Parhami, **Nan Wu**, Sixin Tao, "Taxonomy and Overview of Distributed Malfunction Diagnosis in Networks of Intelligent Nodes", *the CSI Journal on Computer Science and Engineering*, 2016.

**Abstract**

Machine Learning Empowered Agile Hardware Design and Design Automation

by

Nan Wu

With the ever-increasing applications comes the realization that efforts and complexity for developing hardware to keep pace with such compute demands are growing at an even faster rate. And, the problem goes further. As the target cadence of Moore's law is already slipping, more burden is placed on the design methodology to achieve the "equivalent scaling". The proliferation of everywhere machine learning (ML) reveals its multi-faceted role: the killer applications that pull transition to novel hardware and compute paradigms (i.e., system for ML), and the important boosters to design methodology that push toward automated and agile hardware development (i.e., ML for system). Aiming to foster the virtuous cycle between ML and hardware, my research features hardware agile development empowered by ML and studies how to infuse intelligence, improve agility, and eventually enable no-human-in-the-loop automation for scalable and efficacious hardware development flow by synergistic investigation across algorithm, architecture, and electronic design automation (EDA).

Specifically, we investigate how different ML techniques can be applied for (1) fast and accurate design evaluation, (2) efficient and scalable design optimization, and (3) high-quality and productive design verification. In design evaluation, we leverage the inherent graph structures of data flow graphs and circuits and explore how domain knowledge can be infused into graph neural network (GNN)-based models, so that we can reconcile timeliness, accuracy, and generalization capability in high-level synthesis (HLS) and logic synthesis performance predictions. In design optimization, we exploit deep reinforcement

learning for flexible, scalable, and automated design exploration in HLS resource allocation and workload placement optimization, which is efficient in large search spaces and can be transferred to new designs. In design verification, we utilize the message-passing mechanism in GNN computation to imitate conventional symbolic reasoning, which is scalable to extremely large Boolean networks with billions of nodes and makes better use of modern computing resources. Through multiple case studies, we showcase the possibilities and potentials of ML-driven methodologies in agile and intelligent hardware design and design automation. Going forward, we hope to see the virtuous cycle, in which ML-based techniques are efficiently running on the most powerful computers with the pursuit of designing the next generation computers.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

Over the past decade, we all witness the exponentially increasing compute demand in artificial intelligence (AI), which roughly doubles every 3.4 months [1]. By comparison, Moore's law [2, 3], which has been powering integrated circuit revolutions since 1960s, has a two-year doubling period. The discrepancy in scaling trends heralds the implications of developing systems surpassing today's capabilities. As the target cadence of Moore's law is already slipping [4], more burden is placed on the design methodology to achieve the "equivalent scaling" to continue moving to larger-scale, more complex, and heterogeneous designs and systems. However, the explosion of modern hardware complexity is challenging the optimality and scalability of conventional development methodologies and electronic design automation (EDA) tools, resulting in long time-to-market as well as high capital and labor costs: from the time-to-market aspect, nearly 70% of application-specific integrated circuit (ASIC) or field-programmable gate array (FPGA) projects are completed behind schedule in 2020 [5]; from the cost aspect, the development costs of leading-edge electronic designs are skyrocketing [6]; from the tool aspect, existing EDA tools cannot adequately address emerging hardware development [7].

In light of these factors, it seems natural to move towards agile hardware design. The

goal is to expedite the development cycle of next-generation electronic systems while minimizing labor, costs, and design complexity barriers. With a strong desire for increased productivity, it is highly expected to embrace more intelligence in hardware development, prompting a reassessment of the relationship between machine learning (ML) and hardware systems. In addition to optimizing hardware architecture and systems to better support different ML algorithms and applications, recent developments suggest a compelling trend of harnessing ML-based techniques to revolutionize the methodology of hardware design and design automation [8, 9]. This encompasses a twofold meaning: the alleviation of burdens on human experts, and the creation of a virtuous cycle between ML and hardware design.

## 1.1 Interplay of Machine Learning and Hardware

The proliferation of everywhere ML reveals its multi-faceted role: the killer applications that pull transitions to novel hardware and compute paradigms (i.e., system for ML), and the important boosters to design methodology that push towards automated and agile hardware development (i.e., ML for system).

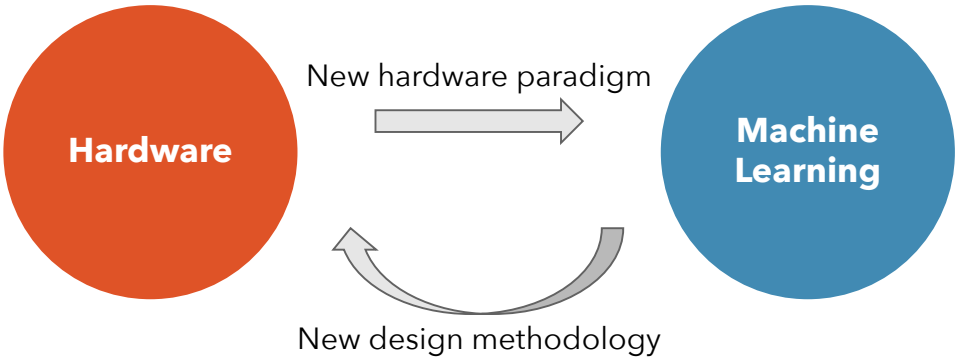


Figure 1.1: In this dynamic landscape, ML plays a dual role, both as the impetus for pushing the boundaries of hardware design and as a powerful tool within hardware design methodologies.

From the system for ML perspective, the primary objective is to enable efficient processing of diverse ML models and applications. This is typically accomplished through either solely hardware design changes (e.g., adopting emerging technologies and exploring architecture/system support for various dataflows [10]) or hardware-software co-design (e.g., reducing the precision or the number of operations and operands) [11].

From the ML for system perspective, ML-based techniques gain widespread popularity in predictive performance modeling and design exploration [8]. In performance prediction, ML-based approaches, especially supervised learning-based methods, often take relatively low-level abstractions of hardware systems as input features to predict various performance metrics or criteria of interest, such as power, energy consumption, latency, and throughput. In design optimization, ML-based methods can either be integrated directly into the design process to facilitate efficient exploration at system, architecture, and micro-architecture levels, or perform run-time control and management at the core, chip, node, and data center levels for scheduling, dynamic power management, and other run-time orchestration and hardware resource management.

## 1.2 Challenges and Opportunities

Challenges often come hand in hand with opportunities. By recognizing the limitations of conventional design approaches and acknowledging the obstacles that may arise in exploiting ML-based techniques for hardware evaluation, optimization, and verification, we can fully unleash the potential of applying ML for more intelligent and agile hardware development flows.

### 1.2.1 Design Evaluation

To accelerate design iterations, it is essential to rapidly and accurately predict the quality and behavior of hardware designs. Traditionally, hardware system performance estimation has relied on analytical models or cycle-accurate simulators. Analytical models often provide fast yet inaccurate predictions [12]. Developing such models entails laborious manual endeavors to comprehend intricate hardware details, leaving them prone to both human and modeling errors. Additionally, their portability is limited, requiring the construction of a new analytical model from scratch for each distinct hardware design. Cycle-accurate and/or instruction set simulators, such as gem5 [13] and sniper [14], provide accurate yet time-consuming estimations. These simulators incur expensive computation costs for performance modeling, thereby restricting scalability to large-scale and complex systems. Moreover, the lengthy simulation time tends to impede the thorough exploration of design spaces.

Compared to conventional modeling methods, a promising alternative lies in data-driven ML-based approaches, with the primary goal to reconcile prediction accuracy, timeliness, and generalization capability. Notably, several challenges should be well addressed, including the collection of sufficient data, enhancing sample efficiency, and effectively transferring knowledge across diverse hardware designs.

### 1.2.2 Design Optimization

From a single processing unit to warehouse-scale computing infrastructures, from application mapping to hardware development, optimization is a pillar stone to achieve various trade-offs among different design specifications, such as performance, energy/power efficiency, and resource utilization. Metaheuristics-based search methods have been extensively applied in design space exploration, such as simulated annealing (SA) [15],

generic algorithms (GA) [16], particle swarm optimization (PSO) [17], and ant colony optimization (ACO) [18]. Despite their effectiveness, they often require initiating the exploration process from scratch for each new design or problem, without leveraging previous experiences or knowledge, which may result in long search time and degraded solution quality. In contrast, ML-based methods possess the capability to accumulate experience with each encountered instance, enabling them to gradually acquire expertise, refine strategies, and hone the generated solutions. They also excel at extracting the underlying relationship between the context and target optimization metrics, which might be implicit to human experts, allowing for decision-making without the need for explicit programming. To thoroughly capitalize on the aptitude of ML-based optimization, it is imperative to investigate effective methods of incorporating domain knowledge into ML models to accelerate the search process, enhance the generalization capability across various designs, and interpret and comprehend the solutions provided by the developed models.

### 1.2.3 Design Verification

Hardware verification has been a bottleneck in chip development, whose scalability is not immune from large-scale designs, system-on-chip (SoC) complexity, heterogeneous integration, the overwhelming volume of verification data, and so on. Two commonly applied verification paradigms are formal verification and simulation-based verification. Formal verification uses static analysis to mathematically prove or disprove the functional correctness of a system with respect to certain formal specifications or properties [19], whose runtime complexity and memory usage hinder its scalability to large designs [20]. Simulation-based verification exercises the designs with valid input stimulus to compare whether the outputs match the oracle provided by a golden reference model [21].

The challenge of scalability extends beyond just the increasing size of designs. Firstly, techniques that are effective at the IP-block-level or sub-system-level verification, such as constrained-random testing and functional coverage analysis, often struggle to scale up to the entire SoC integration or system-level validation. Second, simulation-based verification is producing so many data that it has become a big data problem [22], calling for a new level of intelligence in verification, including but not limited to smarter test generation, coverage collection/analysis, and debug. Reckoning on these challenges, in formal verification, ML-based techniques can enhance the efficiency of satisfiability (SAT) solvers, theorem proving, equivalence checking, and model checking, as well as benefit assertion generation using natural language processing (NLP)-based approaches; in simulation-based verification, ML-based techniques typically serve as predictive models for rapid and precise coverage predictions, automated test generation and selection, as well as troubleshooting assistants.

## 1.3 Contributions and Organization

Aiming to foster the virtuous cycle between ML and hardware, my research features agile hardware development empowered by ML and studies how to infuse intelligence, improve agility, and eventually enable no-human-in-the-loop automation for scalable and efficacious hardware development flow by synergistic investigation across algorithm, architecture, and EDA. Specifically, this dissertation endeavors to demonstrate how ML-based techniques and their effective utilization of modern computing systems benefit (1) fast and accurate design evaluation (Chapter 3 and Chapter 4), (2) efficient and scalable design optimization (Chapter 5 and Chapter 6), and (3) high-quality and productive design verification (Chapter 7), which is organized as follows.

- In Chapter 2, we introduce the background of graph neural networks (GNNs) and

reinforcement learning (RL), which are the primary tools used to improve hardware development process in this dissertation. Additionally, we present a concise literature review of studies that apply ML-based techniques for performance evaluation, design optimization, and hardware verification.

- Chapter 3 seeks to answer why and how GNNs can be applied for high-level synthesis (HLS) performance predictions [23]. We leverage the inherent graph structure of data flow graphs, benchmark 14 state-of-the-art GNN models, and propose a knowledge-infused hierarchical GNN model incorporating domain knowledge to balance prediction accuracy and timeliness. To promote the interdisciplinary research between GNN and HLS, we formally formulate the problem and construct a standard and open-source benchmark suite including a variety of synthetic and realistic applications.

- Chapter 4 innovatively employs multi-modal graph learning to provide accurate assessments of the quality of results (QoR) after logic synthesis [24]. Our approach combines order dependence from sequences of logic transformations (i.e., logic synthesis flows) and structural properties from hardware designs using a hybrid model, which achieves excellent generalization capability across both logic synthesis flows and circuit designs.

- Chapter 5 proposes an end-to-end optimization framework, IronMan [25, 26], for fine-grained, flexible, and automated optimization in HLS, with the primary goal to provide either optimized solutions under user-specified constraints or Pareto trade-offs between different objectives (e.g., resource types and timing). IronMan consists of a code transformer serving as the interface to HLS tools to expose operator-level optimization opportunities, a deep RL-based design space exploration engine

for multi-objective optimization, and a GNN-based performance predictor to enhance knowledge transfer to new applications.

- Chapter 6 focuses on workload placement optimization on multi-chip many-core (MCMC) systems [27]. MCMC systems offer high parallelism via decentralized execution and can scale to very large systems with reasonable fabrication costs. As these systems continue to scale, workload partitioning and placement significantly impact the efficiency of on-chip and off-chip communication. In this context, we propose a deep deterministic policy gradient (DDPG)-based workload mapping method, applicable to systems connected in different topologies and scalable to large systems with thousands of cores.

- Chapter 7 showcases the effective application of graph learning to bolster parallelism and scalability of symbolic reasoning [28], providing wide-ranging benefits in functional verification, logic minimization, datapath synthesis, malicious logic identification, and more. We use the message-passing mechanism in GNN computation to emulate functional propagation and structural hashing in conventional symbolic reasoning methods, allowing for better utilization of modern computing power to promote verification productivity.

- Chapter 8 concludes the dissertation and discusses future research opportunities through advanced ML algorithms, autonomous EDA, and agile hardware development.

# Chapter 2

# Background and Related Work

In this chapter, we first introduce preliminaries of graph neural networks and reinforcement learning. Then, we briefly discuss the related work employing ML-based techniques for performance evaluation, design optimization, and hardware verification, respectively. For more reference, there is a comprehensive survey of applying ML for computer architecture and systems [8].

## 2.1 Graph Neural Network

GNNs have made remarkable strides in representation learning on graph-structured data, such as particle and high energy physics [29, 30], chemical analysis [31, 32], social networks [33, 34], and drug-target predictions [35, 36]. The fundamental GNN model has been recognized as a generalization of convolutions to non-Euclidean data [37], a differentiable variant of belief propagation [38], and an analogy to classic graph isomorphism tests [39]. Regardless of their origins, GNNs typically employ a neural message passing mechanism in which vector messages are exchanged between nodes along the edges in the graph and updated using neural networks.

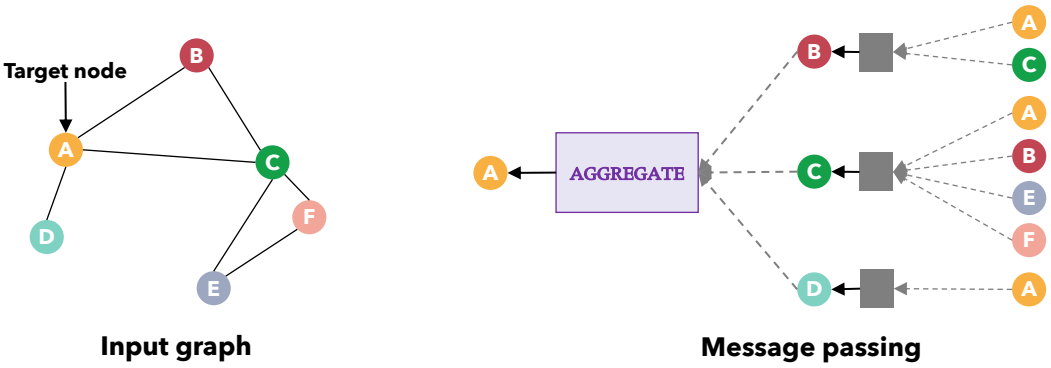


Figure 2.1: Overview of a message-passing model, which depicts how a single node aggregates messages from its local neighborhood.

Given an input graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, each node $v \in \mathcal{V}$ is initialized with a representation $\mathbf{h}_v^{(0)} \in \mathbb{R}^d$, which could be either a direct encoding or a learnable embedding obtained from node features. Then, as shown in Figure 2.1, a GNN layer updates each node embedding by integrating the information from its graph neighborhood $\mathcal{N}(v)$, yielding the representation $\mathbf{h}_v^{(1)}$. This process can be unrolled through time steps by repeatedly using the same update function, deriving representations $\mathbf{h}_v^{(2)}, \mathbf{h}_v^{(3)}, ..., \mathbf{h}_v^{(K)}$. An alternative is to stack several GNN layers, intuitively similar to unrolling through time steps, but increases the GNN capacity by using different parameters in the update function for each time step. This message-passing update can be formally expressed as follows:

$$\begin{aligned} \mathbf{h}_v^{(k+1)} &= \textsc{update}^{(k)}\left(\mathbf{h}_v^{(k)}, \textsc{aggregate}^{(k)}(\{\mathbf{h}_u^{(k)}, \forall u \in \mathcal{N}(v)\})\right) \\ &= \textsc{update}^{(k)}(\mathbf{h}_v^{(k)}, \mathbf{m}_{\mathcal{N}(v)}^{(k)}). \end{aligned} \tag{2.1}$$

Here, UPDATE and AGGREGATE are arbitrary differentiable functions (i.e., neural networks), and $\mathbf{m}_{\mathcal{N}(v)}^{(k)}$ is the "message" aggregated from the neighborhood $\mathcal{N}(v)$ of node $v$. During each iteration, the AGGREGATE function takes in the embeddings of the nodes within the graph neighborhood $\mathcal{N}(v)$ of node $v$ and generates a message $\mathbf{m}_{\mathcal{N}(v)}^{(k)}$; the UPDATE function combines the message $\mathbf{m}_{\mathcal{N}(v)}^{(k)}$ with the previous node embedding $\mathbf{h}_v^{(k)}$ to

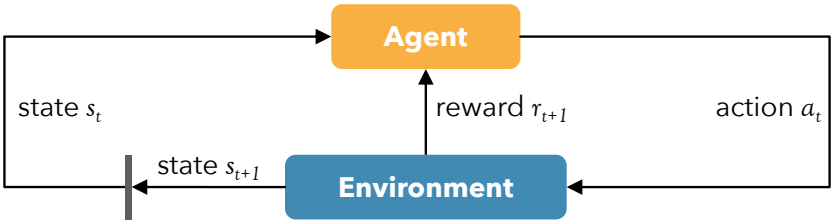


Figure 2.2: A typical framing of RL.

produce the updated embedding.

## 2.2 Reinforcement Learning

RL has achieved great success in many real-world applications, such as AlphaGo [40], self-driving [41], robotics [42], playing video games [43, 44], and financial trading [45, 46, 47]. These instances highlight the versatility and effectiveness of RL in solving complicated decision-making tasks in a wide range of domains.

Figure 2.2 depicts a standard RL scenario [48]. An agent interacts with an environment over a number of discrete time steps. At each time step $t$, the agent receives a state $s_t$ from the state space $\mathcal{S}$, and selects an action $a_t$ from the action space $\mathcal{A}$ according to its policy $\pi$, in which $\pi$ is a mapping from states $s_t$ to actions $a_t$. In return, the agent receives the next state $s_{t+1}$ and a scalar reward $r_{t+1} : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$. This process continues until the agent reaches a terminal state after which the process restarts. The accumulated rewards starting from the time step $t$ can be expressed as:

$$R_{t+1} = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}, \tag{2.2}$$

where $\gamma \in (0, 1]$ is the discount factor. The state-action value $Q_\pi(s, a)$ is represented by

$$Q_\pi(s, a) = \mathbb{E}_\pi[R_{t+1}|s_t = s, a_t = a], \tag{2.3}$$

which is the expected return after selecting action $a$ at state $s$ with policy $\pi$. Similarly, the state value $V_\pi(s)$ is defined as

$$V_\pi(s) = \mathbb{E}_\pi[R_{t+1}|s_t = s], \tag{2.4}$$

which is the expected return starting from state $s$ by following policy $\pi$. The goal of the agent is to maximize the expected return for every state $s$.

There are two general types of RL methods: value-based and policy-based. In value-based methods, the state-action value function $Q_\pi(s, a)$ is approximated by either tabular approaches or function approximations. At each state $s$, the agent always selects the optimal action $a_t^*$ that could bring the maximal state-action value $Q_\pi(s_t, a_t^*)$. One well-known example of value-based methods is Q-learning [49]. As for policy-based methods, they directly parameterize the policy $\pi(a|s;\theta)$ and update the parameters $\theta$ by performing gradient ascent on $\mathbb{E}_\pi[R_{t+1}]$. One example is the REINFORCE algorithm [50], in which the policy parameters $\theta$ are updated in the direction of $\nabla_\theta \log \pi_\theta(s_t, a_t) Q_\pi(s_t, a_t)$.

## 2.3 Related Work on Machine Learning for Performance Evaluation

ML-based techniques can be effectively employed to predict various performance metrics of interest for diverse hardware components, such as memory systems, network-on-chip (NoC), GPUs, and CPUs.

In memory systems, multi-layer perceptron (MLP) [51] and gradient boosting [52] are capable to predict higher-level features (e.g., cache miss, throughput, energy) from lower-level features (e.g, cache configurations). Memory access patterns can be characterized by Block2Vec [53] or GNNs [54], which generate better representations of data blocks,

data flows, and control flows, enabling further optimization for prefetching and branch prediction.

In NoCs, taking buffer/link utilization and NoC configurations as input features, we can apply support vector regression to predict communication latency [55], MLP to predict hotspots [56], ridge regression models to predict energy consumption [57], and decision trees to predict the probability of timing faults [58].

In GPUs, given GPU configurations, performance counters, and kernel characteristics, linear regression can be applied to predict execution time [59, 60]; Wu et al. [61] use clustering and MLPs to model scaling behaviors of general-purpose GPUs (GPGPUs) with respect to the number of compute units, engine frequency, and memory frequency; O'Neal et al. [62] use ensemble learning to predict cross-generation GPU execution time, achieving more than 10,000 times speedup compared to cycle-accurate GPU simulators.

In general processor performance predictions, MLPs and variants of (non-)linear regression are widely utilized to forecast important metrics such as throughput [63], energy [64], power [65, 66], and latency [67, 68], based on micro-architectural configurations and performance counters. These ML-based predictors can be seamlessly integrated to facilitate smart power management. For instance, LEO [69] uses hierarchical Bayesian models to predict performance and power, which is employed to identify the performance-power Pareto frontier for run-time energy optimization; CALOREE [70] decomposes the power management task into two abstractions: a learner responsible for performance modeling, and an adaptive controller leveraging predictions from the learner. These abstractions allow both the learner to use multiple ML techniques and the controller to maintain control-theoretic formal guarantees. Recent studies give more attention on data-driven approaches. For instance, Ithemal [12] leverages a hierarchical multi-scale recurrent neural network (RNN) with long short term memory (LSTM) to predict the throughput of basic blocks. Ding et al. [71] highlight the importance of incorporating

domain knowledge into learning-based modeling, even if the overall accuracy may not be improved. In order to handle data scarcity, they exploit a generative model to generate synthetic training data and apply multi-phase sampling to improve prediction accuracy.

## 2.4 Related Work on Machine Learning for Design Optimization

In the context of design optimization, there are various ways to exploit ML. On the one hand, we can directly develop ML-based components, such as prefetchers and branch predictors. On the other hand, we can incorporate ML-based techniques as part of the design process, such as design space exploration, or use them for run-time control and management.

For ML-based components, we launch discussions on cache replacement/prefetching policies, memory controllers, and branch predictors. To develop better cache replacement and prefetching policies, researchers have explored various ML techniques, such as perceptron learning [72, 73, 74], RL [75, 76, 77], and LSTM [78, 79, 80], which utilize data reuse information and spatio-temporal locality from memory access patterns as well as program semantics to make more intelligent decisions. To devise self-optimizing memory controllers adaptive to dynamic workloads, several studies adopt Q-learning [81, 82, 83], where the memory controller always selects legal DRAM commands with the highest expected long-term performance benefits (i.e., Q-values). To improve dynamic branch predictors, variants of perceptron learning make use of branch histories to decide whether to take the branches [84, 85, 86, 87] and predict the target address of an indirect branch [88].

For incorporating ML-based techniques into the design process, we review several studies that leverage ML-based design exploration in HLS and logic synthesis. Design

space exploration in HLS typically involves the proper assignment of directives (i.e., pragmas) in the high-level source code, since directives have a significant impact on the quality of HLS designs by controlling parallelism, scheduling, and resource usage. Random forest [89] and Bayesian optimization [90] can be used to select suitable loop unrolling factors or optimize the placement of directives (e.g., loop unrolling/pipelining, array partitioning, function inlining, and allocation), with the goal to improve execution latency and resource utilization. In logic synthesis, LSOracle [91] employs an MLP to automatically decide which optimizer should be applied to different parts of circuits; the selection of logic transformations can be optimized by a policy gradient-based method [92].

For ML-based run-time power management, the joint optimization of power gating and dynamic voltage and frequency scaling (DVFS) can be performed by several supervised ML techniques, such as MLP [93], gradient boosting, and k-nearest neighbors [94]. Distributed and multi-level Q-learning approaches can also be applied to select target power modes [95] or conduct DVFS [96, 97]. JouleGuard [98] is a run-time control system coordinating approximate computing applications with system resources under energy budgets. It uses a multi-arm bandit approach to identifying the most energy-efficient system configuration, upon which application configurations are determined to maximize compute accuracy within energy budgets.

## 2.5 Related Work on Machine Learning for Hardware Verification

Simulation-based verification generally consists of three aspects: coverage measurement, test/stimulus generation, and response checking [99]. Accordingly, ML-based techniques usually serve as three roles: predictive models for fast and accurate coverage

predictions, optimizers for test generation or selection, and troubleshooting assistants.

Regarding ML-based predictive models, many studies adopt them as surrogate models to approximate the relationship between tests and coverage metrics, so as to accelerate the verification cycle by fast and accurate coverage predictions without resorting to time-consuming simulation processes for design under verification (DUV). Examples include using ML-based surrogate models for fault coverage [100, 101], code coverage [102, 103], and functional coverage analysis [104].

Regarding ML-based optimizers, it has always been challenging to find the most stressful and comprehensive tests to stimulate DUV so that coverage closure can be reached within time limits. In test generation, after developing surrogate models such as using deep residual neural networks [105] and GNNs [106] to capture the relations between test templates and the expected probabilities of hitting coverage events, a gradient-based search can be applied to propose new tests; RL-based methods, such as policy gradient [107], are also capable of directly generating input stimuli. In test selection, support vector machines (SVMs) [108, 109] and MLP [110] can be used to figure out representative and important tests from a large pool of generated tests and filter out redundant ones.

Regarding troubleshooting assistants, given the large amount of test data, clustering and classification algorithms have showcased their capabilities to automate different steps in debug, such as bug detection with version control systems (VCS) [111], root cause recognition with failure information [112], simulation trace analysis [113], and bug localization with VCS [114] or bug signatures [115].

# Chapter 3

# Hierarchical Graph Neural Network for High-Level Synthesis Performance Prediction

Fast and accurate circuit quality evaluation from early design stages is crucial for agile hardware development. In this chapter, we focus on why and how GNNs can be applied for HLS performance predictions, with the primary goal to reconcile prediction accuracy, timeliness, and generalization capability.

HLS is a prevailing technology to develop ASIC and FPGA designs, which expedites the design flow by automatic transformation from behavioral descriptions in high-level languages (C/C++, etc.) to functionally equivalent RTL designs with different resource/performance trade-offs. Despite the great success achieved by HLS, one of the major challenges is the difficulty in predicting the quality of the generated RTL designs. While invoking the entire synthesis and implementation flow through traditional EDA tools can provide accurate performance evaluation, it is usually extremely time-consuming. Even though HLS tools provide performance estimations, they are far

from accurate [116, 117]. Thus, many efforts strive for accurate performance predictions without invoking the time-consuming implementation process. Classic approaches use analytical models [118, 119, 120], which typically work well for highly regular dataflows such as perfect loops and arrays. Existing ML-based approaches rely on intensive feature extraction and empirical feature engineering from HLS reports or intermediate results of a partially executed implementation process [121, 122, 123], which still requires the core synthesis process that may take minutes to hours as well as high domain expertise.

Motivated by such limitations and the strong desire for agile hardware development, we aim to approach HLS performance predictions *at the earliest design stage with the least feature engineering*, i.e., right after front-end compilation. Since the compiled programs are often represented as intermediate representation (IR) graphs, we exploit the representation power of GNNs on these graphical data to rapidly and accurately predict post-implementation performance metrics. Our contribution is summarized as follows.

- **Benchmarking.** We build a standard benchmark suite with 40k C programs, including synthetic programs and three sets of real-world HLS benchmarks. Each program is compiled to derive the IR graph, synthesized by HLS tools, and implemented on an FPGA device to obtain post place-and-route performance metrics as the ground truth.

- **Modeling.** We formally formulate the HLS performance prediction problem on IR graphs and profile 14 state-of-the-art GNN models. To investigate the trade-offs between prediction accuracy and timeliness (i.e., early or late stages in HLS), we propose two approaches directly using GNNs: (1) *off-the-shelf approach* at the earliest stage with the least domain-specific information; (2) *knowledge-rich approach* at a later stage with HLS auxiliary information to improve prediction accuracy.

- **Advancing.** We further propose a *knowledge-infused approach* using a novel hier-

archical GNN, which decouples the complicated prediction tasks into two simpler ones: node-level classification to infuse domain knowledge and graph-level regression to estimate overall resource usage and timing. This approach reaps timely predictions as well as ample domain knowledge, largely improving prediction accuracy with almost zero overhead at inference time.

We extensively evaluate our proposed predictors on both synthetic and *unseen* real-world programs, which outperform HLS tools by up to 40× in terms of resource usage and timing predictions. The benchmark and explored GNN models are publicly available at https://github.com/lydiawunan/HLS-Perf-Prediction-with-GNNs.

This chapter is organized as follows: Chapter 3.1 discusses the performance prediction strategies used in existing ML-based methods and highlights the strengths of our three proposed prediction strategies; Chapter 3.2 introduces the developed benchmark suite; Chapter 3.3 presents the details of the three proposed approaches; Chapter 3.4 includes the evaluation and discussions on the off-the-shelf, knowledge-rich, and knowledge-infused approaches; Chapter 3.5 concludes this chapter.

## 3.1 Performance Prediction Strategies

For HLS performance predictions, there are two fundamental questions to answer: ***when*** and ***how***.

### 3.1.1 When to Predict

Circuit quality can be predicted at different stages of the design flow, such as before or after HLS, or during implementation. HLS tools take in behavioral descriptions in high-level languages, convert them to RTL, and produce a synthesis report, which

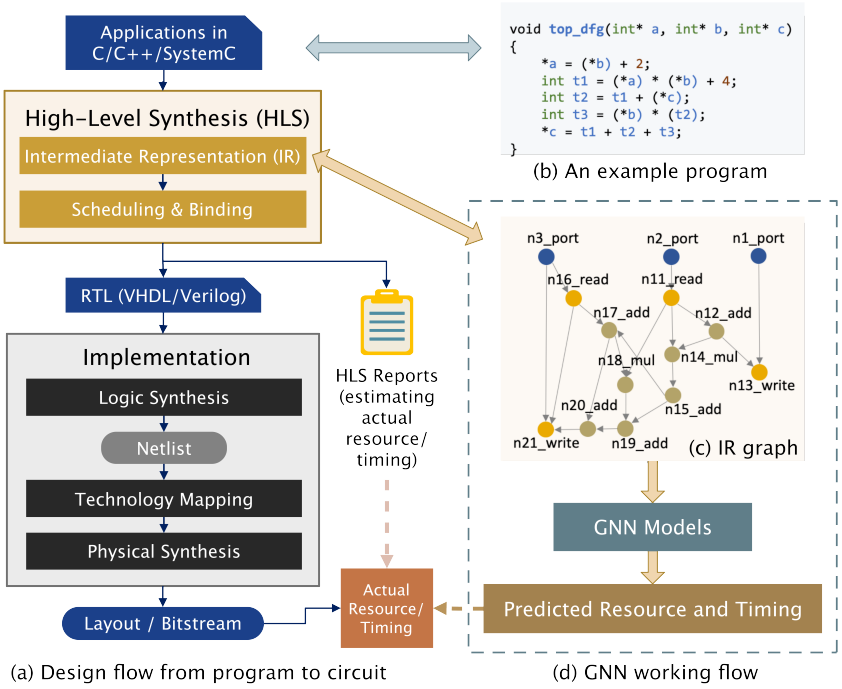


Figure 3.1: The overall performance prediction flow. (a) Design flow starting from behavioral programs to hardware circuits. (b) An example program written in C. (c) The IR graph extracted after front-end compilation. (d) The working flow of GNNs, predicting *actual* resource usage and timing merely based on raw IR graphs.

provides an early estimation of the final performance, as shown in Figure 3.1. However, even after minutes or hours of synthesis, these estimations provided in synthesis reports can be largely inaccurate [116, 117]. Among the existing ML-based predictors for latency, throughput, power, and resource utilization, there are three major sources for feature extraction: HLS directives [124, 125, 126], IRs after HLS front-end compilation [127, 128, 117], and HLS reports [116, 121, 122, 129]. In general, an early and timely prediction benefits agile development, but little domain-specific knowledge is exposed at this stage, which probably hurts prediction accuracy. Thus, there awaits a comprehensive comparison among prediction strategies at different HLS stages in terms of prediction accuracy and timeliness.

### 3.1.2 How to Predict

Existing ML-based prediction approaches attempt linear regression, MLP, SVM, random forest, and ensemble models [116, 121, 122, 123, 124, 125, 126, 127, 128, 129]. Although promising, these models require heavy feature engineering to provide sufficient features as model inputs. For instance, Dai et al. [116] leverage 87 features from HLS reports; Pyramid [121] and XPPE [122] require up to 183 features; HLSPredict [123] relies on 75 features. These features can only be obtained by *actually running HLS or CPU/FPGA sub-trace generation*. Another concern with these methods is their limited generalization capability. For instance, Koeplinger et al. [127] adopt pre-characterized area models to prepare inputs to their MLP-based predictor; several studies take the directives in HLS scripts as input features, but they are design-specific [124, 125]. These indicate that a new model must be trained when encountering a new design. Given that existing ML-based methods require either re-running HLS or re-training a new model to handle every new design, it is highly expected to have more advanced methods with strong generalization capabilities across designs for HLS performance predictions.

### 3.1.3 Our Prediction Strategy

Our goal is to assist agile hardware development by making the performance prediction *as early as possible* and *as accurate as possible*. Specifically, we focus on predicting the *implemented*, i.e., post place-and-route, design performance metrics on FPGA devices, including resource utilization and critical path (CP) timing, without invoking HLS or implementation processes.

To address existing limitations (i.e., late prediction and weak generalizability), we propose three early prediction approaches that leverage varying amounts of domain-specific knowledge, as illustrated in Figure 3.2. To make it *timely*, we perform predictions

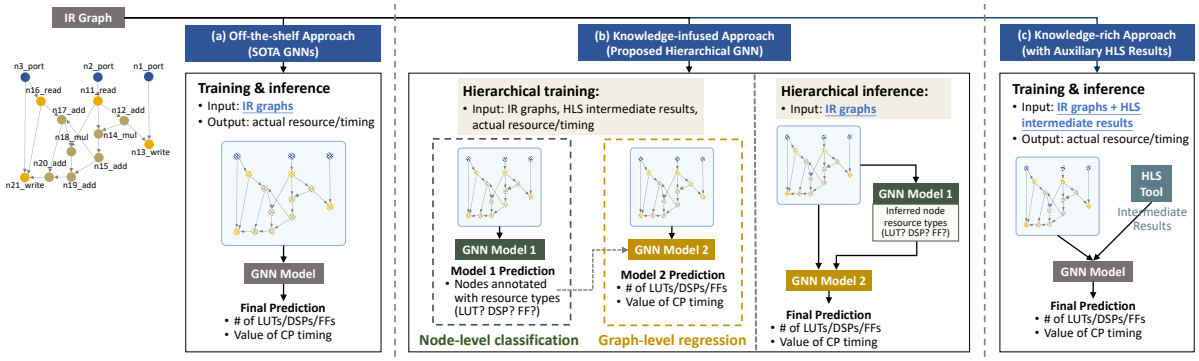


Figure 3.2: Three proposed approaches: (a) off-the-shelf approach at the earliest stage for prediction; (b) knowledge-infused approach, also at the earliest stage but with self-inferred domain-specific information; (c) knowledge-rich approach after partial execution of HLS to obtain auxiliary information.

based on the IR graph of a program, i.e., data flow graph (DFG) and control data flow graph (CDFG), which can be quickly extracted after the front-end compilation [130] within seconds. To make it *generalizable*, we propose to apply GNNs on DFGs/CDFGs, exploiting the inductiveness of GNNs to make accurate predictions for completely unseen designs without retraining. Specifically, the three approaches are:

- *Off-the-shelf approach.* The first approach directly predicts post-implementation performance metrics at the earliest stage by taking IR graphs (DFG/CDFG) as inputs to GNN models, as depicted in Figure 3.2 (a). This approach extracts features immediately after HLS front-end compilation, resulting in the fastest prediction, but with compromised accuracy due to the ignorance of hardware-specific information.

- *Knowledge-rich approach.* As shown in Figure 3.2 (c), the second approach draws support from auxiliary domain information distilled from intermediate HLS results (i.e., partial execution of HLS but no implementation): the resource usage associated with each node. Armed with rich domain knowledge, this approach emphasizes more on prediction accuracy, especially for resource estimation, yet compromises

timeliness and efficiency since HLS tools do take some time to generate intermediate results.

- *Knowledge-infused approach.* As shown in Figure 3.2 (b), the third approach is a *hierarchical GNN-based* prediction strategy that reaps the advantages of the previous two approaches: not only does it make the earliest prediction but also benefits from domain knowledge with almost zero overhead during inference. The knowledge infusion is achieved by decoupling the complicated prediction task into two steps: the first step is a node-level classification for resource types, in which the domain knowledge is infused during training and can be self-inferred during inference; the second step is graph-level regression that estimates the numerical resource usage and timing on top of the node-level inference.

## 3.2 Benchmarking

To facilitate interdisciplinary research in GNN and HLS, we build a standard benchmark suite to promote rapid circuit performance prediction, which includes plenty of synthesizable programs together with *actual* (i.e., post-implementation) performance metrics on FPGA devices.

### 3.2.1 Benchmark Format

We outline the benchmark format in terms of the following components: input graphs, prediction tasks and labels, and node and edge features.

### Input

An IR is a data structure used by compilers to represent the source code of a program, which serves as a high-level abstraction of the program's semantics, enabling further processing such as optimization and translation [130, 131]. In HLS, DFGs and CDFGs are the most common types of IR graphs, which can be quickly extracted after the front-end compilation Therefore, we utilize the IR graph of a program as the input to our GNN-based predictor. Specifically, DFGs are the graphs translated from *basic blocks*, a straight-line code sequence with no branches in except to the entry and no branches out except at the exit [132]; CDFGs are the graphs translated from programs with loops, jumps, and branches. The main difference between DFGs and CDFGs is that DFGs do not include any loops, while CDFGs include additional nodes and edges/loops to represent control dependencies.

### Prediction Task and Label

We provide two types of tasks, a *node-level classification* task, and a *graph-level regression* task, where the former is easier than the later.

- For the *node-level classification* task, each node in IR graphs is assigned a label indicating the resource type(s) that the node will use in its final implementation. We consider three resource categories: `DSP`, `LUT`, and `FF`. A node can be implemented by zero, one, or multiple types of resource. For example, a `sdiv` node may use both `DSP` and `LUT`; a `partselect` node only uses `FF`; a control-related node may use nothing, e.g., `br` that indicates a branch entry. We organize the resource type prediction as three binary classification tasks.

- For the *graph-level regression* task, each IR graph is labeled with corresponding post-implementation performance metrics. We consider four metrics for regression:

Table 3.1: Prediction tasks of resource usage and timing on FPGAs.

| Resource and Timing | Description |
|---|---|
| `DSP`: Digital Signal Processor | A small processor able to quickly perform mathematical operation on streaming digital signals. |
| `FF`: Flip-Flop | A small memory component able to store a bit, typically used as a fast register to store data. |
| `LUT`: Look-Up Table | A set of logic gates hard-wired on FPGAs, storing predefined truth tables and performing logic functions. |
| `CP`: Critical Path Timing | The maximum signal delay of a path from an input to an output, usually in the unit of nanoseconds. |

`DSP`, `FF`, `LUT`, and `CP`. As summarized in Table 3.1, the first three are integer numbers indicating the number of resources used in final implementation; the last one is `CP` timing slack in fractional number, determining the maximum working frequency of an FPGA device.

### Node and Edge Features

The three proposed approaches use different sets of node features, as listed in Table 3.2. After HLS front-end compilation, there are seven node features immediately available for the off-the-shelf approach, such as node category, bitwidth, and opcode. For the knowledge-infused and the knowledge-rich approaches, we additionally include the resource type and the number of used resources in node features, respectively. Notably, for the knowledge-infused approach, the auxiliary node features (i.e., resource type) are *only* provided during training, while during inference they are inferred by our node-level GNN. Each edge has two features, the edge type represented in integers, and a binary value indicating whether this edge is a back edge.

Table 3.2: Node features and example values.

| Feature | Description | Values |
|---|---|---|
| Off-the-shelf approach with minimum domain information | | |
| Node type | General node type | `operation nodes`, `blocks`, `ports`, `misc` |
| Bitwidth | Bitwidth of the node | `0`∼`256`, `misc` |
| Opcode type | Opcode categories based on LLVM | `binary_unary`, `bitwise`, `memory`, etc. |
| Opcode | Opcode of the node | `load`, `add`, `xor`, `icmp`, etc. |
| Is start of path | Whether the node is the starting node of a path | `0`, `1`, `misc` |
| Cluster group | Cluster number of the node | `-1`∼`256`, `misc` |
| Knowledge-infused and knowledge-rich approach | | |
| `DSP` | `DSP` used for this node? | binary/integer values, `misc` |
| `LUT` | `LUT` used for this node? | binary/integer values, `misc` |
| `FF` | `FF` used for this node? | binary/integer values, `misc` |

### 3.2.2 Benchmark Generation

We construct the synthesizable benchmark suite including synthetic C programs as well as real-world HLS applications. The synthetic programs fall into two categories, basic blocks that derive DFGs, and programs with control loops and branches that derive CDFGs. All of the synthetic programs are generated by the C program generator `ldrgen` [133], a variant of Frama-C [134]. For graph-level tasks, there are 19,120 and 18,570 C programs in the DFG and CDFG datasets, respectively. The node-level dataset contains more than 660k nodes derived from DFGs and CDFGs. In addition, there are three sets of real-world HLS applications: MachSuite [135], CHStone [136], and PolyBench/C [137], consisting 16, 10, and 30 different applications, respectively. The real-world applications are *only* used for generalization evaluation of GNN models and we do not include them during training. More statistics of the benchmark suite is analyzed in Program-to-Circuit [138].

## 3.3 Modeling and Advancing with GNNs

We provide more details of the three proposed GNN-based approaches with various trade-offs between timeliness and accuracy. GNNs operate by propagating information along the edges of a given graph, allowing each node to receive and update its own embedding based on its neighboring nodes. By stacking multiple GNN layers, each node can receive information from multi-hop neighbors and locally characterize the corresponding receptive field for node-level tasks. Graph pooling is then used to summarize global information and perform graph-level prediction tasks.

### 3.3.1 Modeling: Off-the-Shelf Approach with State-of-the-Art GNN Models

In the off-the-shelf approach, we screen several state-of-the-art GNN models, aiming to identify (1) which properties of existing GNN models would help with resource/timing prediction and (2) how domain-specific insights can be combined with these properties to improve prediction accuracy. 14 different GNN models are selected from four categories based on how topological and relational information in graphs are exploited. We briefly introduce them as follows.

**Graph Convolutional Network (GCN) and variants**

- GCN [139] is inspired by the first order graph Laplacian methods, which essentially performs aggregation and transformation on node representations without learning trainable filters.
- GCN can be equipped with a virtual node [32]; this virtual node serves as a global scratch space that each node reads from and writes to in every step of message

passing.

- SGC [140] is a simplified version of GCN, which reduces computation complexity through successively removing nonlinearities and collapsing weight matrices between consecutive layers, corresponding to a fixed low-pass filter followed by a linear classifier.
- GraphSage [39] can be recognized as a variant of GCN, which samples a fixed number of neighbor nodes to keep the computational footprint consistent.
- The convolution operation using auto-regressive moving average filters (ARMA) [141] can offer a larger variety of frequency responses and can account for higher-order neighborhoods compared to polynomial filters with the same number of parameters.
- PAN [142] considers path integral information in the convolution operation, which is a generalization of GCN that assigns trainable weights to each path depending on its length.

**Graph Isomorphism Network (GIN) and variants**

- GIN [143] is provably as powerful as Weisfeiler-Lehman graph isomorphism test, due to the use of sum aggregators over a countable input feature space.
- GIN can also be equipped with a virtual node [32].
- Principle neighborhood aggregation (PNA) [144] emphasizes the necessity to use complementary aggregators, which allows each node to better understand the graph structure and retain neighborhood information, especially under a continuous input feature space. The sum aggregator is generalized as a combination of a mean aggregation and degree-scalers, enabling the network to amplify or attenuate signals based on the degree of each node.

**Employment of Multi-Relational Information**

- Graph attention networks (GATs) [145] apply attention mechanisms to implicitly assign different importances to nodes in the same neighborhood.
- Gated graph neural networks (GGNNs) [146] have trainable edge-dependent weights with gated recurrent units.
- Rather than unrolling layer-wise computation through time steps as GAT and GGNN, relational GCN (RGCN) [147] utilizes edge-dependent weights with non-linear activations, which is specifically designed to characterize multi-relational data and contextual information.

**Inspired from Vision Tasks**

- Inspired by the advances in pixel-wise prediction tasks brought by encoder-decoder architectures such as the U-Net, graph U-Net [148] develops an encoder-decoder structure on graph, which can encode and decode high-level features while maintaining local spatial information.
- Inspired by the feature-wise linear modulation (FiLM) in the visual question answering domain, GNN-FiLM [149] makes use of hypernetworks in learning on graphs, combining learned message-passing functions with dynamically computed element-wise affine transformations.

To fairly evaluate these models, we use the same GNN structure (e.g., same embedding size, same layer count) but with different types of GNN layers. The goal is to directly predict actual resource/timing based on IR graphs without invoking HLS. This off-the-shelf approach makes the earliest predictions since HLS front-end compilation is the very first step of an EDA design flow. While with the best timeliness, the accuracy is

compromised due to the ignorance of hardware-specific information. For example, the GNN model does not have the information of what resources will be used to implement a certain node (e.g., `LUT` or `DSP`?), making it hard to accurately predict the total resource utilization of entire IR graphs.

### 3.3.2 Modeling: Knowledge-Rich Approach with Selected GNN Models

To incorporate more hardware-specific information revealed during the synthesis flow, we devise the knowledge-rich approach, which takes both IR graphs and auxiliary information from intermediate HLS results as inputs, as shown in Figure 3.2(c). The auxiliary information from HLS tools indicates both the resource type(s) and the number of each type of resource used in the final implementation for every node in IR graphs. As each node is marked with pre-characterized resource estimations, the GNN model can pay more attention to resource interference/sharing among nodes, achieving much better prediction accuracy.

Equipped with rich domain knowledge, this approach emphasizes more on prediction accuracy, especially for resource estimation, yet compromises timeliness and efficiency, since HLS tools generally consume minutes to hours to generate intermediate results. Evaluation of this approach is conducted using the top-performing GNN models identified from the screening in the off-the-shelf approach.

### 3.3.3 Advancing: Knowledge-Infused Approach with Hierarchical GNN Models

To strike a balance between timeliness and accuracy, we propose the knowledge-infused approach with hierarchical GNN models. As depicted in Figure 3.2(b), the re-

source/timing prediction is disentangled into two tasks: a node-level classification task that annotates resource types associated with each node, and a graph-level regression task that predicts actual resource/timing with the annotated graphs. We use two separate GNN models for these two tasks, and adopt hierarchical training and inference.

- **Hierarchical training**. First, the node-level classification GNN takes IR graphs as inputs, and the domain knowledge is infused by providing labels to each node that denote resource types used in final implementation based on HLS intermediate results. Second, the graph-level regression GNN then takes both IR graphs and ground-truth resource types as inputs, aiming to convey the infused domain knowledge from node-level to graph-level tasks and to improve final prediction accuracy.
- **Hierarchical inference**. During inference, the only inputs required for the two trained GNNs are the IR graphs. First, the node-level GNN model infers resource types for each node. Second, combining the node-level inference results with original IR graphs, the graph-level regression GNN grasps self-inferred domain knowledge to perform final predictions.

Taking advantages of knowledge infusion during training, this approach demonstrates a great balance between timeliness and accuracy: predicting resource/timing from the earliest stage and simultaneously adopting adequate domain information to improve prediction accuracy.

## 3.4 Experiment

### 3.4.1 Experimental Setup

All GNN models are implemented with Pytorch Geometric [150]. The ground-truth (actual) resource usage (`LUT`/`DSP`/`FF`) and `CP` timing are synthesized by Vitis HLS [151]

and implemented by Vivado [152]. DFG and CDFG datasets are randomly split into 80% train, 10% validation and 10% test; real-world benchmarks are only used for generalization evaluation. Each GNN model is empirically set as five layers with a hidden-dimension size of 300. For graph-level regression, sum or mean pooling is used to derive graph representations, followed by a feed-forward network with the structure 300-600-300-1. Models are trained using Adam optimizer for 100 epochs. Learning rates, dropout and other hyper-parameters are tuned on the validation set. Each model is trained with five runs using different random number seeds and we report the average of three with least validation error.

### 3.4.2 Modeling: SOTA GNN Analysis

We launch discussions of the off-the-shelf approach from three aspects: (1) how different graph structures influence prediction accuracy; (2) which properties of existing GNN models would help improve accuracy; (3) what domain-specific insights can be derived to facilitate future graph representation learning on fast and generalizable evaluation in EDA tasks.

#### Different Graphs: DFG vs. CDFG

. Table 3.3 exhibits the mean absolute percentage error (MAPE) of predictions on DFGs and CDFGs from synthetic programs. The MAPE on CDFGs is larger than that on DFGs, which attributes to two major reasons. First, DFGs do not have loops but CDFGs typically include a considerable number of loops [138]. Since message-passing-based GNN models have limited expressiveness and are not better than the 1-Weisfeiler-Lehman isomorphism test [153], they are not excelled to handle graphs with many loops. Second, control signals introduce additional nodes/edges that represent control states and de-

Table 3.3: MAPE of graph-level regression with different GNN models on DFG and CDFG datasets. The two top-performing models are marked in bold.

| | DFG | | | | CDFG | | | |
|---|---|---|---|---|---|---|---|---|
| | DSP | LUT | FF | CP | DSP | LUT | FF | CP |
| **GCN** | 16.31% | 16.49% | 21.27% | **6.12%** | 25.30% | 28.64% | 38.34% | 8.79% |
| **GCN-V** | 15.72% | 15.93% | 21.64% | 6.36% | 17.31% | 33.93% | 39.94% | **8.13%** |
| **SGC** | 42.12% | 23.93% | 30.61% | 7.92% | 44.01% | 60.87% | 53.50% | 10.32% |
| **SAGE** | 15.18% | 14.01% | 17.11% | **6.12%** | 17.01% | 28.09% | 39.11% | **8.25%** |
| **ARMA** | 19.12% | 13.46% | 16.87% | 6.50% | 18.47% | **25.21%** | 32.15% | 8.42% |
| **PAN** | 15.24% | 14.13% | 17.23% | 6.38% | 16.88% | 32.65% | 44.36% | 8.54% |
| **GIN** | 15.52% | 16.10% | 22.08% | 6.58% | 15.47% | 28.48% | 38.82% | 8.76% |
| **GIN-V** | 15.04% | 16.17% | 23.09% | 6.40% | 17.94% | 29.40% | 48.64% | 8.59% |
| **PNA** | **12.65%** | **11.64%** | **14.41%** | 6.26% | **14.71%** | **22.86%** | **26.47%** | 8.87% |
| **GAT** | 26.22% | 22.64% | 27.74% | 8.30% | 28.66% | 46.19% | 54.73% | 10.32% |
| **GGNN** | 15.40% | 13.64% | 16.94% | 6.47% | 16.28% | 28.05% | 31.88% | 8.50% |
| **RGCN** | **13.27%** | 13.03% | **15.09%** | 6.14% | **15.03%** | 26.33% | **25.52%** | 8.72% |
| **UNet** | 18.40% | 14.90% | 19.17% | 6.61% | 18.92% | 32.83% | 53.06% | 9.02% |
| **FiLM** | 20.05% | **12.50%** | 16.94% | 6.27% | 17.42% | 26.97% | 27.35% | 8.67% |

pendency, which are seemingly unrelated to resource usage but can easily confuse GNN models during resource prediction; meanwhile, control signals are usually accompanied with more complex memory operations [154], such as `store` and `alloca`, further complicating the allocation of `FF` and `LUT`.

### GNN Model Analysis

. PNA and RGCN generally show superior performance, implying two takeaways. First, the relational information (i.e., edge information) is important in IR graphs, since it represents data or control dependency, or a mix of both, which is a critical basis in logic synthesis and impacts resource allocation. Second, equipped with multiple aggregators, PNA is more powerful to characterize different neighborhood information, thus providing better prediction accuracy.

**Domain-Specific Insights**

. Among three types of resource, DSPs are mainly used for computation; FFs often relate to memory operations and small arrays; LUTs may appear in computation, memory or control nodes. The key to making precise DSP prediction is to distinguish major computation nodes that are most likely to use DSPs. For instance, a multiplication node with a large bitwidth tends to use DSPs, while divisions and bitwise operations prefer LUTs. Similarly, effective extraction of memory-related nodes would greatly benefit FF predictions. Since LUTs are involved in the entire graph (as computation units and glue logic to circuit components), graph-level understanding is important. To briefly summarize, it is helpful to carefully characterize neighborhood information from each node's predecessors, successors, itself, and their relations, such that the sophisticated mapping rules from heterogeneous nodes to resource usage can be clearly understood and quantitatively learned.

Compared with resource predictions, CP timing predictions show relatively lower MAPE and better consistency between DFGs and CDFGs. A probable reason is that CP timing is local information and thus is insensitive to graph sizes as long as the critical path segment can be recognized.

### 3.4.3 Advancing: Comparison of Three Approaches

We first discuss the results of the knowledge-infused approach, and then comprehensively compare the three proposed approaches with commercial HLS tools.

**Evaluation on Knowledge-Infused Approach**

Essentially, using GNNs to predict actual resource/timing from IR graphs is to approximate the set of sophisticated heuristics and mapping rules used by HLS schedul-

Table 3.4: Prediction accuracy of node-level resource classification with four different GNN models on DFGs, CDFGs and real-case applications.

| | DFG | | | CDFG | | | Real Case | | |
|---|---|---|---|---|---|---|---|---|---|
| | **DSP** | **LUT** | **FF** | **DSP** | **LUT** | **FF** | **DSP** | **LUT** | **FF** |
| **GCN** | 93.79% | 84.84% | 88.66% | 83.00% | 77.01% | 64.74% | 79.70% | 81.83% | 86.82% |
| **SAGE** | 93.06% | **87.32%** | **92.09%** | 85.65% | 78.41% | 60.40% | 87.39% | 86.44% | 55.88% |
| **GIN** | 93.80% | 84.93% | 91.57% | 79.24% | 73.05% | 65.78% | 74.70% | 75.53% | 72.24% |
| **RGCN** | **93.91%** | 87.13% | 91.52% | **85.80%** | **78.46%** | **68.92%** | **90.82%** | **88.83%** | **91.55%** |

Table 3.5: MAPE of the three proposed approaches with RGCN/PNA on DFG and CDFG datasets. The default notation means the off-the-shelf approach; -I means the knowledge-infused approach; -R means the knowledge-rich approach.

| | DFG | | | | CDFG | | | |
|---|---|---|---|---|---|---|---|---|
| | **DSP** | **LUT** | **FF** | **CP** | **DSP** | **LUT** | **FF** | **CP** |
| **RGCN** | 13.27% | 13.03% | 15.09% | 6.14% | 15.03% | 26.33% | 25.52% | 8.72% |
| **RGCN-I** | 10.60% | 10.25% | 12.47% | 5.70% | 12.65% | 20.55% | 19.01% | 6.78% |
| **RGCN-R** | 8.86% | 8.58% | 10.18% | **4.91%** | 10.98% | 14.06% | 16.65% | **5.46%** |
| **PNA** | 12.65% | 11.64% | 14.41% | 6.26% | 14.71% | 22.86% | 26.47% | 8.87% |
| **PNA-I** | 8.26% | 5.10% | 7.58% | 5.51% | 10.39% | 14.12% | 16.42% | 6.54% |
| **PNA-R** | **7.06%** | **4.02%** | **5.78%** | 5.39% | **8.95%** | **10.27%** | **11.22%** | 5.81% |

ing/binding and logic/physical synthesis during the design flow. The evaluation of the off-the-shelf approach in Table 3.3 indicates that *plug-in application of GNNs cannot well approximate such underlying rules*. Thus, in addition to infusing domain knowledge during training, another motivation of the hierarchical structure in the knowledge infused approach is to *divide and conquer*. The complicated performance prediction is decoupled as two simpler tasks that are solved separately. For *node-level classification*, Table 3.4 shows prediction accuracy of classifying resource types, where high accuracy is achieved for most of the cases since local neighborhood characterization is enough for node-level resource type classification. For *graph-level regression*, Table 3.5 displays MAPE of predictions on synthetic programs, showing an accuracy boost compared with the off-the-shelf approach.

With the hierarchical training, both the node-level and the graph-level GNN models in Figure 3.2(b) are approximating simplified design heuristics. Specifically, the node-level classification aims to understand the preference of resource types on different nodes; the graph-level regression focuses on globally estimating resource sharing and interference among nodes. With the hierarchical inference, the domain knowledge infused during training can be self-inferred when encountering unseen designs, leading to improved prediction accuracy from the earliest design stage.

**Reconciling Prediction Accuracy, Timeliness, and Generalization Capability**

The three approaches explore different trade-offs between timeliness and accuracy. Intuitively, the more domain information is leveraged, the more accurate predictions are provided, whereas the longer time would be taken for feature collection. The off-the-shelf approach makes predictions from the earliest stage simply with IR graphs, at the cost of accuracy loss due to the ignorance of domain knowledge. The knowledge-rich approach provides the best prediction accuracy, but has to wait for HLS tools providing intermediate results, sacrificing timeliness. The knowledge-infused approach shows a balance: infusing adequate domain knowledge during training, and making predictions from the earliest stage during inference.

Generelization capability is a key indicator of whether an ML- or GNN-based approach can be widely applied for certain EDA tasks. Table 3.6 shows the MAPE of the three proposed approaches and Vitis HLS on real-world applications. Compared with Vitis HLS, our approaches significantly improve the prediction accuracy, especially for `LUT`/`FF` usage and `CP` timing. Specifically, PNA-based knowledge-infused approach outperforms HLS by 1.2× to 40.6×, and PNA-based knowledge-rich approach outperforms HLS by 1.7× to 51.4×.

Such results empirically demonstrate (1) the generalization capability not only from

Table 3.6: Testing MAPE of the three proposed approaches with RGCN/PNA on real-case applications.

| | HLS | RGCN | RGCN-I | RGCN-R | PNA | PNA-I | PNA-R |
|---|---|---|---|---|---|---|---|
| **DSP** | 26.07% | 45.61% | 40.89% | 32.90% | 40.06% | 21.95% | **15.20%** |
| **LUT** | 871.56% | 66.23% | 30.91% | 24.08% | 56.34% | 21.45% | **16.96%** |
| **FF** | 322.86% | 101.20% | 38.75% | 27.72% | 47.65% | 20.10% | **17.42%** |
| **CP** | 32.09% | 8.13% | 5.35% | 5.83% | 8.68% | 4.80% | **3.97%** |

seen to unseen designs but also from synthetic to realistic applications, and (2) accuracy and timeliness conspicuously surpassing HLS tools.

## 3.5 Conclusion

In this chapter, we discuss three approaches for early circuit performance prediction using GNNs: (1) the off-the-shelf approach, which makes the earliest prediction with least domain-specific information, showing on-par performance with HLS; (2) the knowledge-rich approach, which makes late prediction after HLS with auxiliary information, showing significantly better performance than HLS; (3) the knowledge-infused approach, which makes the earliest prediction in a two-step hierarchical manner with self-inferred knowledge, still significantly outperforming HLS. We also construct a standard benchmark suite to facilitate future research. This chapter not only demonstrates the great potential of applying GNNs for HLS performance predictions, but also advances the GNN design by proposing innovative architectures.

# Chapter 4

# Multi-modal Graph Learning for Logic Synthesis QoR Prediction

In this chapter, we investigate the potential of multi-modal graph representation learning to predict the quality of results (QoR) after logic synthesis. Our approach takes advantage of both the structural information from circuit designs and the temporal (i.e. relative ordering) information from logic synthesis flows, achieving high prediction accuracy and better generalization capability.

Industrial investigations [155] have highlighted two fundamental requirements for production-ready ML in EDA: (1) ML-based performance estimations should achieve a minimum of $2\sigma$ accuracy ($\sim$95%); (2) the developed ML model should possess strong generalization capability, allowing it to directly apply to new designs without re-training.

To this end, we target logic synthesis and propose a novel approach for highly accurate QoR estimations with great generalization capability, as shown in Figure 4.1. We emphasize the importance of utilizing the **spatio-temporal information** to forecast QoR (i.e., delay/area). Specifically, the structural characteristics inside hardware designs are distilled and represented by GNNs; the temporal knowledge (i.e., the relative

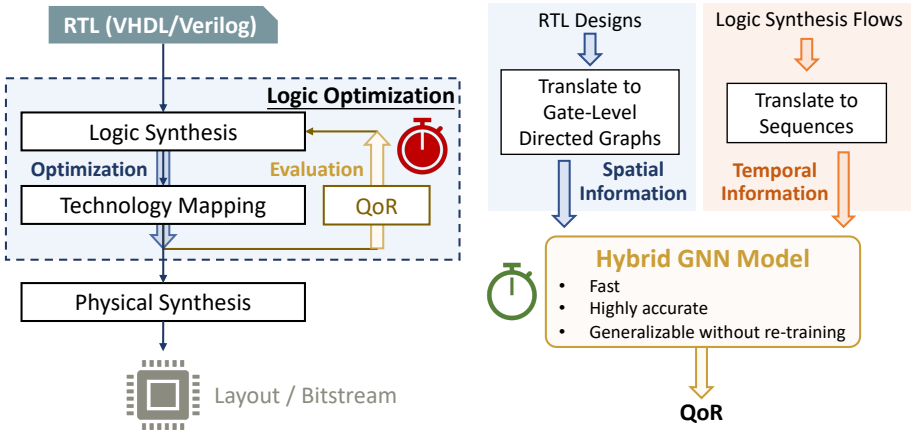


Figure 4.1: The design flow and the proposed approach to predicting QoR after applying logic synthesis flows on hardware designs. (a) The focus of this chapter is to accelerate the evaluation phase in logic optimization. (b) The proposed model exploits spatial information from circuit designs and temporal knowledge from logic synthesis flows, generalizable to new designs without re-training.

ordering of logic transformations) in logic synthesis flows can be imposed on hardware designs by combining either a virtually added supernode or a sequence processing model with conventional GNN models. We summarize our contributions as follows.

- **Modeling**. We propose two generalizable GNN-based approaches to predicting QoR of logic synthesis flows by incorporating the crucial spatio-temporal information from both hardware designs and synthesis flows. To capture the impact of synthesis flows on circuits, the first approach utilizes a supernode on GNN, and the second approach combines LSTM and GNN in a hybrid manner. In particular, the second approach *represents the graphs and synthesis flows separately*, reducing training complexity and memory overhead; it also better represents the problem nature, where circuits (represented by graphs) and synthesis flows (represented by sequence) are two separate concepts.

- **Evaluation**. Evaluations on designs seen and unseen during training show the

superiority of our approach. On the seen designs, i.e., the transductive scenario, the MAPE achieved by the hybrid GNN is less than 1.2%, 7× lower than existing studies. On the unseen designs, i.e., the inductive scenario, the MAPE is still below 3.2%, 14× lower than existing studies.

- **Dataset**. We provide an open-source dataset consisting of 3.3 million data points collected from eleven different circuit designs, with the goal to facilitate multi-modal or dynamic graph representation learning for EDA tasks. Our dataset and ML models are publicly available at https://github.com/lydiawunan/LOSTIN.

This chapter is organized as follows: Chapter 4.1 provides the motivation, related work in logic optimization, and preliminaries of supernode and LSTM; Chapter 4.2 details the problem formulation and the proposed hybrid GNN models that exploit spatio-temporal information; Chapter 4.3 evaluates the proposed models in comparison to existing ML-based methods; Chapter 4.4 concludes this chapter.

## 4.1 Motivation, Related Work, and Preliminaries

We begin by presenting our motivations, followed by a summary of related work that employs ML-based methods for logic optimization. Finally, we provide a brief introduction to supernode and LSTM.

### 4.1.1 Motivation

Logic synthesis is a process of transforming RTL designs into optimized logic-gate-level representations. A logic synthesis flow refers to a sequence of logic optimizations, and a well-designed flow can significantly reduce design area and latency. In spite of

decades of research, there remain unresolved challenges and requirements for efficient logic optimization, as follows.

- The design space of possible logic synthesis flows is extremely large [156, 157], which reemphasizes the importance of fast and accurate QoR prediction for sufficient design space exploration.

- There is no one-for-all solution. Commercial EDA tools usually provide reference design flows [158] developed by experts based on heuristics or prior knowledge, but such flows do not uniformly perform well. As shown in Figure 4.2(a), first, for a specific circuit design, different flows have drastically varied optimization effects; second, the same set of flows have different performance across different designs. These observations suggest the importance of **design-specific** synthesis flows.

- The transformation order in synthesis flows should be well captured. Figure 4.2(b) compares the impact of different flow lengths, where the distribution of area or delay is not conspicuously improved with longer flows. This indicates that it is the underlying **temporal information**, i.e., the relative ordering of logic transformations, inside synthesis flows that majorly determines final QoR.

- Existing approaches lack generalization capability across different designs. Prior studies utilize convolution neural network (CNN) [156] or LSTM [157] to predict QoR for a certain design. These methods target fixed-length flows and have limited generalization capability to unseen designs due to the absence of design-specific information as model inputs. Aiming at a practical use of ML-based performance modeling, the generalization across different designs and flow lengths is a necessity.

Reckoning on the aforementioned issues, we propose an innovative solution by employing multi-modal graph learning to leverage the spatio-temporal information from

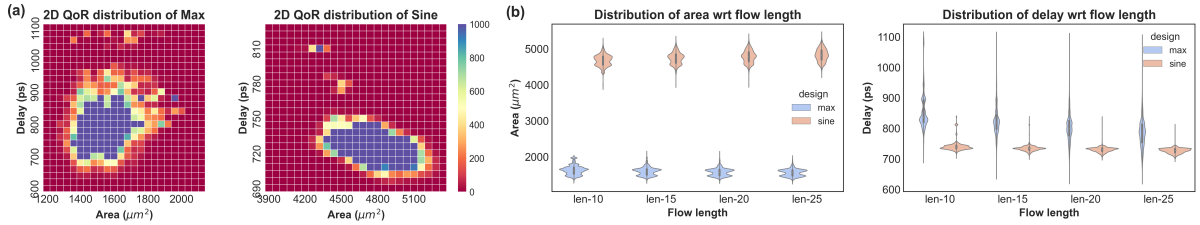


Figure 4.2: Area and delay results of 300,000 random logic synthesis flows applied on circuit designs `max` and `sin`, respectively. The number of count no less than 1,000 is represented by the same color.

hardware designs and logic synthesis flows, enabling accurate and generalizable QoR predictions.

### 4.1.2 Related Work

In logic optimization, the sequence to apply logic transformations, i.e., the logic synthesis flow, is often determined heuristically. For example, commercial EDA tools provide reference synthesis flows [158]; an academia open-source logic synthesis tool ABC [159] offers synthesis flows *resyn*, *resyn2* and *resyn2rs*.

Recently, ML-assisted logic optimization has attracted increasing research interests, aiming to reduce exploration time and improve performance. For example, LSOracle [91] employs MLP to automatically decide which one of the two optimizers should be applied on different parts of circuits. The logic optimization can also be formulated as an RL problem, implemented with a GNN-based agent [92, 160] or a non-graph based agent [161]. The optimization objective is to minimize area [92, 160, 161] or delay [92]. In terms of forecasting logic synthesis flow performance, a convolution neural network (CNN) [156] can be used to identify whether a synthesis flow is an angel-flow or a devil-flow; LSTM [157] can be applied to predict delay and area after applying a synthesis flow.

From a broader view, GNNs are expected to make better use of graph structured data

in many EDA problems [162]. Instead of conventional graph representation learning that maps circuit designs from static graphs to labels (e.g., resource/timing/power) [8, 163], the target task in this work should consider both circuit designs (i.e., static graphs) and synthesis flows (i.e., transformations to be applied on the graphs) to provide high-accuracy predictions of delay and area, which can be recognized as *a multi-modal or dynamic graph representation learning*.

### 4.1.3 Preliminaries

We briefly describe the two techniques that will be used to capture the temporal information inside logic synthesis flows.

#### Supernode in GNNs

The introduction of a supernode aims to address the difficulty in propagating information across remote parts of graphs [32, 164]. The supernode is a newly added virtual node that connects all the nodes in the original graph to promote global information propagation by reducing the maximum distance between any two nodes to two hops. Many GNN models can be equipped with such a supernode, which serves as a global scratch space that every other node reads from and writes to in every step of message passing with some preference.

#### LSTM

LSTM [165] is a type of RNNs capable of learning the order dependence and long-term dependence in sequence processing problems. For each unit in a sequence, the same computations are performed and the current output states are dependant on the previous (hidden) states. Theoretically, LSTM can process sequential inputs such as sentences in

arbitrary length. A common LSTM unit consists of a cell, an input gate, an output gate, and a forget gate, among which the cell is responsible to remember values over arbitrary time intervals and the rest three gates regulate the information fed into and out of the cell. Given its sequential information processing capability, an LSTM-based model is a proper candidate to represent logic synthesis flows.

## 4.2 Proposed Hybrid GNN Models

We propose a novel ML approach that is fast, accurate, and generalizable for estimating QoR of logic synthesis flows by leveraging spatio-temporal information. Two models are explored: (1) a GNN for spatial information learning, armed with a supernode to encode temporal information (Chapter 4.2.2); (2) a hybrid model, composed of a GNN for spatial learning and an LSTM for temporal learning (Chapter 4.2.3).

### 4.2.1 Problem Formulation

In logic synthesis, hardware designs are converted to logic networks, which are typically graph abstractions of logic circuit implementations in the gate level. Logic optimization aims to manipulate and transform logic networks to reduce the amount of required hardware or the critical path delay by sequences of logic transformations, which are referred to as logic synthesis flows.

#### Prediction Task

We leverage ABC [159], an open-source logic synthesis framework well-adopted in academia, to generate synthesis flows. Notably, any other logic synthesis tool can be used in place of ABC as long as sufficient training data are available. The inputs to the proposed predictors are initial hardware designs described in RTL and the logic synthesis

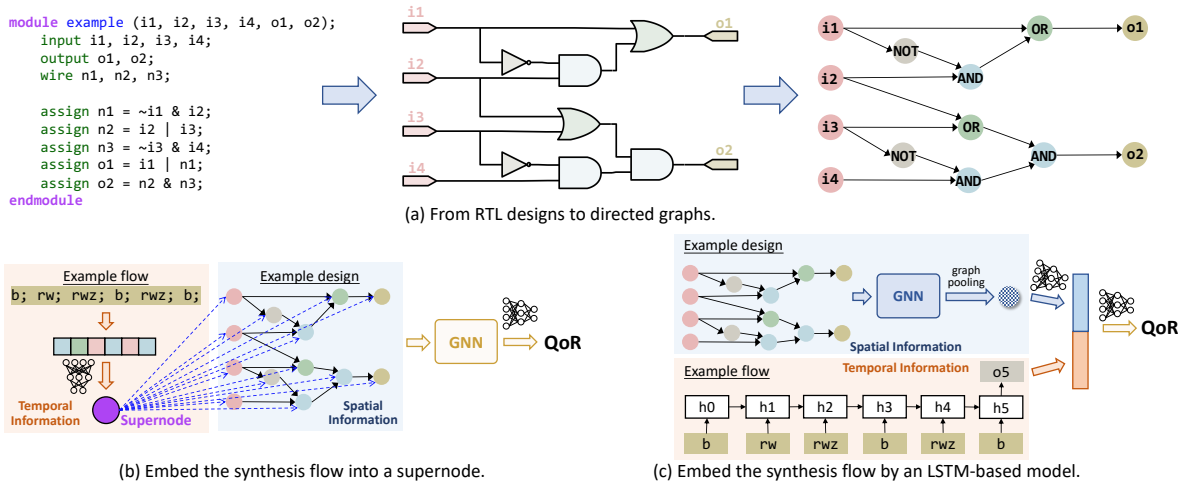


Figure 4.3: The overview of our proposed GNN architectures. (a) Logic synthesis takes in register-transfer-level (RTL) descriptions and converts to gate-level netlists, from which we build directed graphs. (b) The proposed GNN with supernode. (c) The proposed hybrid GNN with LSTM.

flows to be applied. The QoR metrics to be predicted are the logic area (denoted as area) and the critical path delay (denoted as delay), and the ground truth is collected from ABC after technology mapping. This prediction task can be extended to other flow performance estimation problems, such as the resource utilization in HLS [23] and negative slacks in placement and routing (i.e., physical synthesis) [166, 167].

### Graph Representation for Circuits

As logic optimization targets gate-level transformations, we represent circuit designs as directed graphs, where each node is a primary logic gate and each edge shows logic dependency. A Verilog parser is built to translate RTL designs into gate-level netlists. To guarantee universal representations of *any combinational logic functions*, `AND`, `OR`, and `NOT` gates are included in the translated graphs. Multiple-output or more-than-two-input gates are automatically split and parsed into the three types of gates aforementioned. Thus, each node has two attributes: (1) node type in input/intermediate/output, and (2) operation type in `AND`/`OR`/`NOT`. Such a parser enables the circuit representations to be

independent of logic optimizers targeting different logic representations [91] (e.g., And-Inverter Graphs (AIGs) [159] and Majority-Inverter Graphs (MIGs) [168]), fostering the portability across different logic synthesis tools. The process of transforming an RTL design into a directed graph is exemplified in Figure 4.3(a).

**Flow Representation**

Within the ABC framework, we consider logic synthesis flows composed of 7 types of logic transformations from $\mathbb{S} = \{$*balance (b), resubstitution (rs), resubstitution -z (rsz), rewrite (rw), rewrite -z (rwz), refactor (rf), refactor -z (rfz)*$\}$. To integrate the inherent temporal information from synthesis flows with circuit designs, a synthesis flow can be represented as either (1) a vector to construct a supernode that directly propagates temporal knowledge to circuit designs (Chapter 4.2.2) or (2) a sequence embedding generated by an LSTM model (Chapter 4.2.3).

**Efficiency of Separate Representations for Circuits and Sequences**

One straightforward way is to merge a circuit design with one particular synthesis flow to generate a single graph (where the flow becomes node attributes), and directly apply vanilla GNN models to produce unified representations. However, this approach has two major issues. First, by plugging the flow into node attributes, all the nodes will share similar node representations, resulting in over-redundant input features. Second, it is nearly impossible to build a graph dataset for graph-level regression, in which each graph is large-scale and each node has many node attributes. Current graph representation learning pays attention to either graph-level tasks on relatively small graphs [153] or node/link-level tasks on large graphs [33]. Generating datasets that involve a large number of large graphs along with copious node attributes can cause out-of-memory issues, impractical for implementation and vulnerable to scalability. In our preliminary experi-

ments, such a unified representation would generate a dataset over 80 gigabytes, which is challenging for efficient training. Thus, we propose to represent the circuit designs and synthesis flows separately. During training, the graph representation does not need to be repeatedly loaded for different synthesis flows, which significantly reduces the memory overhead for storage and training as well as the training time by orders of magnitude.

### 4.2.2 GNN with Supernode

Inspired by the idea that introducing a supernode to graphs can collect and redistribute global information with some preference [32, 164], we propose to leverage a supernode to represent synthesis flows. Since the supernode is virtually connected to all nodes in the original graph, the temporal information is directly injected into the circuit graph, as shown in Figure 4.3(b).

#### Supernode to Embed Synthesis Flows

Synthesis flows are converted to fixed-length input vectors with the dimension of 25, since the maximum length of currently considered synthesis flows is 25. Each logic transformation in a flow is represented as an integer from 1 to 7, and zero padding is applied for flows shorter than 25. In Figure 4.3(b), the initial input vector of the example flow is encoded as [1, 6, 7, 1, 7, 1, 0, ...]. A single fully-connected layer then converts the $1 \times 25$ vector into $1 \times 8$ as the supernode embedding.

#### Spatial Representation of Circuit Structures

To study the impact of synthesis flows on a circuit design, we connect the supernode to all other nodes in the original graph. The two node attributes are converted to learnable node embeddings as $1 \times 8$ vectors. The modified graphs are passed through GNN models

for graph representation learning. By exposing the temporal information encoded in the supernode and distributing it to the entire graph, this model is expected to process both spatial and temporal features simultaneously, i.e., learn the effects of synthesis flows on different circuit structures.

### 4.2.3 Hybrid GNN with Spatio-Temporal Information

While a supernode is capable to collect global information and distribute temporal knowledge to every other node in graphs, we notice three concerns that may influence prediction performance. First, synthesis flows are represented by fixed-length vectors, which are then passed through an MLP to comply with the embedding dimension of other nodes. This setting is insensitive to sequence dependence, i.e., the transformation order in synthesis flows, whereas the impact of later transformations heavily depends on previous ones. Second, as the message-passing process proceeds, the original temporal information inside the supernode is gradually faded in other nodes. Third, by adding the supernode that connects all the nodes in the original graph, the graph size increases with newly added edges, which may cause scalability issue in implementation when encountering extremely large graphs.

To address the first concern, a more natural way is to leverage a sequence processing model to distill the temporal information, and the specific model employed in this work is LSTM, which excels at handling order dependence and variable-length flows. To resolve the second and the third concerns, we separately generate a sequence embedding (i.e., a synthesis flow representation) and a graph embedding (i.e., a circuit representation) in the feature extraction stage, and these two embeddings are concatenated for downstream predictions. The approach of learning two separate embeddings and then concatenating them is intuitively consistent with the actual logic synthesis procedure, as it mirrors the

process of applying the synthesis flow to the circuit.

Figure 4.3(c) illustrates the structure of the second hybrid GNN model with LSTM. The directed graph translated from a circuit design is passed through a GNN model, followed by a linear layer to generate a graph-level representation (i.e., a $1 \times 32$ vector). The synthesis flow is processed by a two-layer LSTM to derive a flow representation (i.e., a $1 \times 64$ vector). These two vectors are concatenated to form a $1 \times 96$ vector. Finally, a feed-forward MLP with the structure of 96-100-100-1 is adopted for delay and area predictions.

## 4.3 Experiment

We first describe the dataset generation and the setup for our experiments, and then present our evaluations with discussions on the results.

### 4.3.1 Dataset Generation

As shown in Table 4.1, we select eleven circuit designs from the EPFL benchmark [169], a benchmark suite designed as a comparative standard for logic optimization and synthesis. The logic synthesis flows are generated by the logic synthesis tool ABC [159]. To demonstrate the flexibility of handling variable-length synthesis flows, we create synthesis flows consisted of 10, 15, 20, and 25 logic transformations; for each different length, there are 50,000, 50,000, 100,000, and 100,000 different flows, respectively, totally making up 300,000 flows. Each synthesis flow consists of logic transformations from $\mathbb{S} = \{b,\ rs,\ rsz,\ rw,\ rwz,\ rf,\ rfz\}$. All the 300,000 flows are applied to eleven different designs with ASAP 7nm low-voltage technologies [170], which are 3.3 million data points in total. The ground truth (i.e., label) is collected from ABC after technology mapping.

Table 4.1: Graph size of different circuit designs.

| | Number of Nodes | Number of Edges |
|---|---|---|
| **adder** | 2926 | 3690 |
| **arbiter** | 24258 | 35841 |
| **bar** | 6935 | 10136 |
| **div** | 143375 | 200494 |
| **log2** | 68881 | 100909 |
| **max** | 6752 | 9105 |
| **multiplier** | 59404 | 86338 |
| **sin** | 11486 | 16878 |
| **sqrt** | 57296 | 81786 |
| **square** | 42042 | 60460 |
| **voter** | 35105 | 47862 |

### 4.3.2 Experimental Setup

#### Baseline

The proposed hybrid GNN models are compared against two existing ML-based approaches: a CNN-based model [156], and an LSTM-based model [157]. We exactly follow the model structures mentioned in the prior studies, with minor modifications to fit our prediction task.

- In the CNN baseline, each transformation is represented as a one-hot vector; synthesis flows (with the maximum length of 25) are represented as $7 \times 25$ matrices with zero padding for shorter flows. We honor the original CNN structure [156], in which there are two convolutional layers followed by a max-pooing and two fully connected (FC) layers. Since the CNN-based model was designed for binary classification, we replace the final classifier by a single neuron for numeric predictions in our task. Note that the CNN-based model can only be trained in a *design-specific* manner, i.e., one model for one design.

- In the LSTM baseline [157], we replace the one-hot embeddings of transformations

with learnable embeddings. To make the model design-agnostic, we add design names as the prefix to synthesis flows to construct new sequences, and intentionally train one model for all designs to study its generalizability. This model consists of two LSTM layers with a hidden size of 128, followed by an MLP of 128-30-30-1 to predict delay and area.

**Implementation and Training**

All the aforementioned neural network models are implemented with Pytorch [171] and Pytorch Geometric [150]. Experiments were performed on a Linux host with a 64-core Intel Xeon Gold 5218 CPU (2.30 GHz) and Nvidia RTX 2080Ti GPUs.

Training, validation, and testing sets are split by 20:5:75. We highlight two training and evaluation strategies. First, in contrast to many ML tasks that use a large proportion of the entire dataset for training, we intentionally train the proposed models with a small portion and conduct evaluations on the rest data points. Second, we evaluate both **transductive and inductive** scenarios. If a design is encountered during training but with new flows during testing, it is considered a transductive scenario. On the other hand, if a design is completely unseen during training and only used for testing, it is referred to as an inductive scenario. The goal is to emphasize the generalizability of our proposed models, which is important for many EDA tasks that are possibly suffering from data scarcity. Among eleven circuit designs, six of them (`adder`, `arbiter`, `bar`, `div`, `log2`, and `max`) are used for both training and testing, and the remaining five (`multiplier`, `sin`, `sqrt`, `square`, and `voter`) are **merely evaluated in testing to demonstrate generalization, i.e., inductive capability**. Training details are summarized as follows.

- For each design, we train a design-specific CNN for 20 epochs with the RMSprop optimizer (learning rate 0.05).

- LSTM is trained for 100 epochs with the Adam optimizer (initial learning rate 2e-3, weight decay 2e-6).
- For the GNN with supernode (denoted as GNN-S), a ten-layer GIN model is trained for 20 epochs with the Adam optimizer (learning rate 1e-3); node and edge embedding dimensions are 8 and 2, respectively.
- For the hybrid GNN-LSTM model (denoted as GNN-H), a ten-layer GIN is combined with a two-layer LSTM (whose hidden size is 64), trained for 20 epochs with the Adam optimizer (initial learning rate 2e-3, weight decay 2e-6). The node embedding dimension is 32.

### 4.3.3 Evaluation

#### Transductive Scenario

Table 4.2 shows the MAPE of QoR predictions on the designs that are seen during training but with unseen synthesis flows. We have the following observations. (1) Since the CNN baseline is design-specific, it slightly outperforms the LSTM-based model, which is a unified model across all designs. (2) The hybrid GNN model, GNN-H, significantly outperforms the LSTM-based model, with 7× and 15× lower MAPE than those of area and delay predictions, respectively. (3) The GNN with supernode, GNN-S, shows comparable performance with the LSTM-based model.

#### Inductive Scenario

Table 4.3 shows the MAPE of QoR predictions on unseen designs. (1) The CNN-based model only works for design-specific synthesis flows and thus there is no generalization to unseen designs. (2) The LSTM-based model suffers from a large accuracy degradation

Table 4.2: Comparison with CNN and LSTM in the **transductive** scenario. **GNN-S** is the proposed GNN with supernode; **GNN-H** is the proposed hybrid GNN.

| | Area (MAPE) | | | | Delay (MAPE) | | | |
|---|---|---|---|---|---|---|---|---|
| | **CNN** | **LSTM** | **GNN-S** | **GNN-H** | **CNN** | **LSTM** | **GNN-S** | **GNN-H** |
| **adder** | 7.00% | 8.72% | 7.65% | 0.87% | 1.76% | 16.22% | 1.79% | 0.76% |
| **arbiter** | 2.98% | 13.66% | 8.16% | 1.56% | 0.23% | 18.96% | 15.37% | 1.86% |
| **bar** | 8.46% | 5.22% | 22.72% | 1.61% | 0.74% | 14.98% | 22.59% | 2.06% |
| **div** | 12.71% | 7.75% | 13.16% | 0.88% | 7.72% | 14.31% | 9.29% | 0.16% |
| **log2** | 8.04% | 9.05% | 4.19% | 0.55% | 3.87% | 11.85% | 12.74% | 0.53% |
| **max** | 7.28% | 6.18% | 8.35% | 1.48% | 5.50% | 17.37% | 20.60% | 0.68% |
| **MEAN** | 7.75% | 8.43% | 10.70% | 1.16% | 3.30% | 15.62% | 13.73% | 1.00% |

Table 4.3: Comparison with LSTM in the **inductive** scenario.

| | Area (MAPE) | | | Delay (MAPE) | | |
|---|---|---|---|---|---|---|
| | **LSTM** | **GNN-S** | **GNN-H** | **LSTM** | **GNN-S** | **GNN-H** |
| **multiplier** | 57.82% | 9.39% | 2.45% | 38.21% | 17.89% | 1.75% |
| **sin** | 66.09% | 64.48% | 2.34% | 45.94% | 54.44% | 2.32% |
| **sqrt** | 29.03% | 39.25% | 4.83% | 38.03% | 15.75% | 2.09% |
| **square** | 38.59% | 13.96% | 2.86% | 47.52% | 31.34% | 2.41% |
| **voter** | 27.38% | 76.49% | 3.08% | 42.19% | 46.54% | 0.96% |
| **MEAN** | 43.78% | 40.71% | 3.11% | 42.38% | 33.20% | 1.91% |

for unseen designs, indicating limited generalization capability. (3) The GNN-S slightly outperforms the LSTM-based model by 3% and 9% in area and delay predictions, respectively (further discussed in Chapter 4.3.4). (4) The GNN-H maintains its high prediction accuracy by slightly increasing the MAPE from 1% to 3%, demonstrating extraordinary generalization capability.

### Sensitivity Analysis

We study the design choices of GNN-H in terms of GNN types and the number of layers. Figure 4.4, 4.5, 4.6, and 4.7 compare the MAPE of QoR predictions with respect to both GIN and GCN models with different number of layers. Generally, GIN

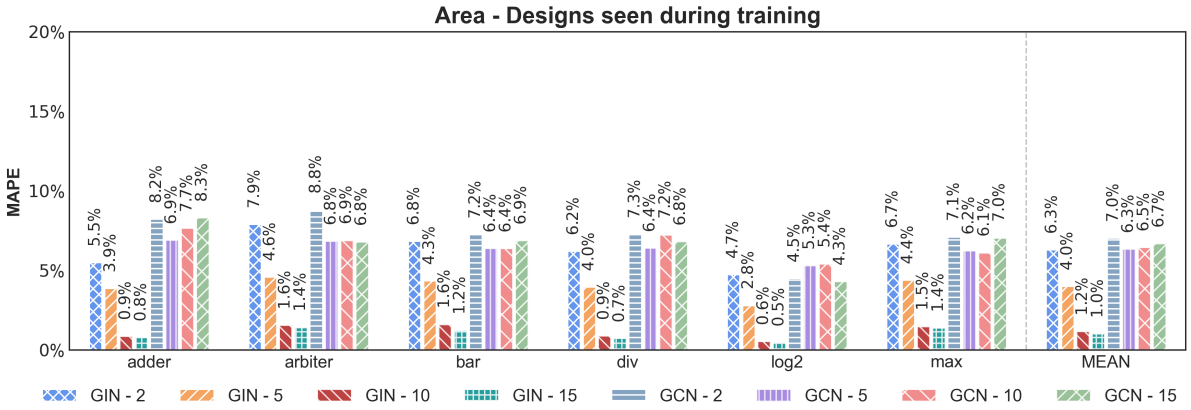


Figure 4.4: Transductive MAPE on area predictions made by GNN-H. Results are compared in terms of GIN and GCN with different number of layers.

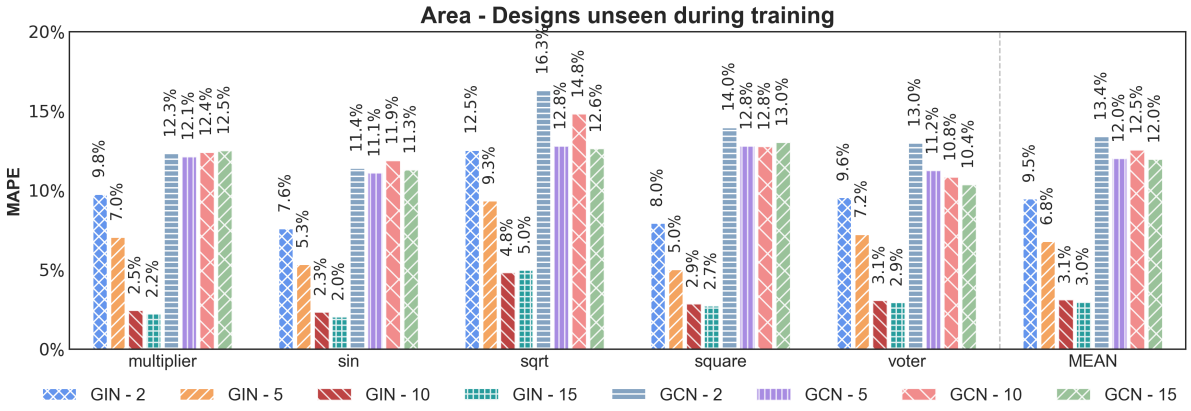


Figure 4.5: Inductive MAPE on area predictions made by GNN-H. Results are compared in terms of GIN and GCN with different number of layers.

models receive an accuracy boost after stacking ten layers, whereas GCN models show similar prediction accuracy among different choices of layers. (1) Regarding the GNN type comparison, GCN suffers from the over-smoothing problem [172]. Mathematically, GCN [139] is an approximate of $2I_N - L$, where $L$ is the normalized graph Laplacian operator and $I_N$ is the identity matrix. Since the graph Laplacian operator/filter is a high-pass filter, GCN naturally becomes a low-pass filter, indicating that stacking many layers does not help to better capture graph structures. (2) Regarding the number of GNN layers, a deep GNN setting with carefully selected GNN types possesses better

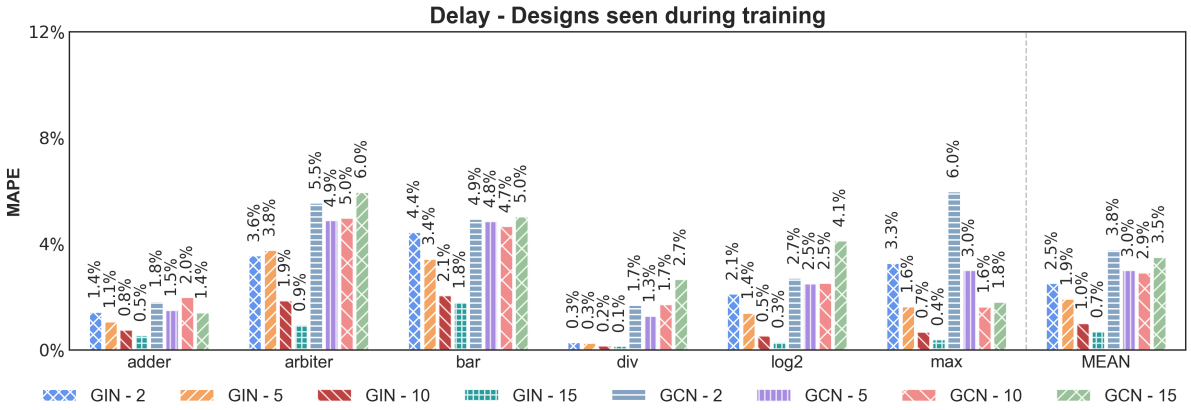


Figure 4.6: Transductive MAPE on delay predictions made by GNN-H. Results are compared in terms of GIN and GCN with different number of layers.

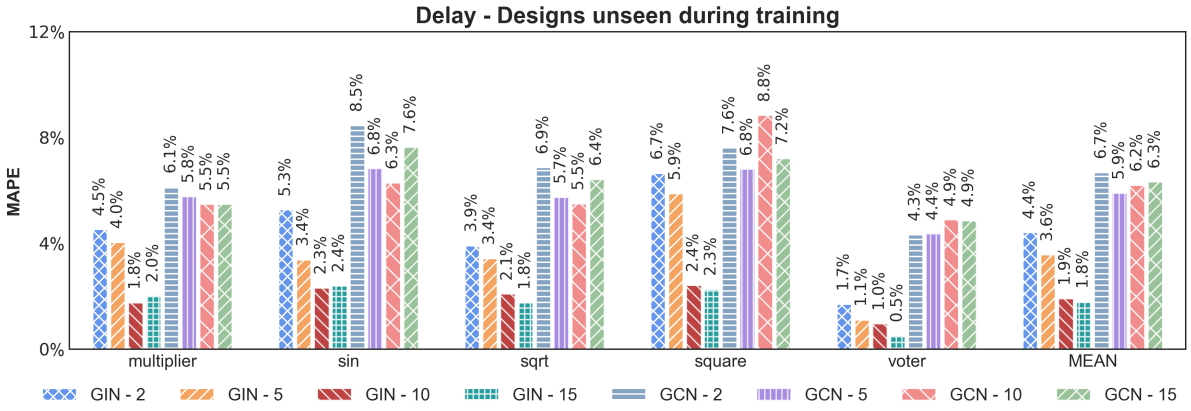


Figure 4.7: Inductive MAPE on delay predictions made by GNN-H. Results are compared in terms of GIN and GCN with different number of layers.

representation power, since stacking more layers enlarges the receptive field to better characterize input graphs, especially beneficial for large graphs.

### 4.3.4 Discussion and Insight

#### GNN-S v.s. GNN-H

In GNN-S, even though a synthesis flow is encoded as a supernode, there are several limitations that influence temporal information characterization. First, every synthesis

flow is directly represented as a fixed-length vector to generate a supernode embedding, which is insensitive to sequence dependence, i.e., the order of logic transformations. Second, the original temporal information injected to the supernode is gradually diluted, since the supernode embedding also evolves during the message passing. We compared different communication mechanisms between the supernode and other nodes, i.e., bidirectional or unidirectional, where the prediction accuracy is slightly improved with the unidirectional communication. This demonstrates that the slower the dilution rate is, the more temporal knowledge can be reserved during learning. Third, simply adding a supernode into original graphs may not be an efficient approach to fusing information from different modalities (i.e., graph-structured data and sequence-structured data). By contrast, GNN-H leverages a more direct scheme that combines the strengths of both GNN and LSTM to extract spatio-temporal information in a decoupled manner: the LSTM directly characterizes temporal information from synthesis flows, and the GNN focuses on representing spatial structures of circuit designs. Rather than GNN-S that mixes spatial and temporal information at the very first step, separately learned graph and sequence embeddings have better expressiveness for each input modality, thus providing a better foundation for downstream tasks.

**Scalability Regarding Graph Abstraction Level**

Table 4.1 shows the gate-level graph size of different circuit designs. The bright side is both GNN-S and GNN-H can handle large graphs. The dark side is the graph size will explode for larger circuit designs, which may cause scalability issues in practical implementation. GNN-S may exhibit some scalability concerns due to the considerable number of virtual edges added to the original graphs, i.e., $|V|$ virtual edges will be additionally added for a graph original with $|V|$ nodes. Two potential directions to further improve scalability are (1) extracting graphs from higher level of circuit abstractions to

provide graphs of appropriate sizes that can be easily handled by both GNN-S and GNN-H, or (2) hierarchically clustering nodes in gate-level graphs [173] to ensure reasonable compute costs for each stage.

### Multi-Modality Graph Representation Learning

Graph representation learning has evolved from single-modal to multi-modal [174], with several attempts of exploiting multiple modalities (visual, acoustic, textual) in videos for personalized recommendation [175] and multi-modality biomarkers for accurate diagnosis of Alzheimer's disease [176]. By contrast, there is a stagnation in the EDA domain: prior studies that adopt GNNs for fast evaluation focus on mapping static circuit graphs to metrics of interest [163, 8]. Thus, the most significant innovation of this work stems from two aspects. First, we consider input information from multiple modalities, i.e., circuit designs in graph format and synthesis flows in sequence format, since the final QoR of circuit designs is dependent on both circuit structures and synthesis flows. Our investigation with GNN-S and GNN-H shows that efficient approaches to extracting features and fusing information from different modalities can conspicuously improve representation power. Second, we build a large dataset to provide initial efforts on facilitating multi-modality graph learning for circuit designs. The multi-modal graph representation learning, which integrates the knowledge from other learning schemes with the conventional graph representation learning, is expected to provide more versatility for EDA tasks.

### Generalization to Other Transformations

Even though the main focus of this work is the generalization across different circuit designs, which is in fact a more practical case as synthesis tools usually hold a fixed set of transformations awaiting to be applied on different circuit designs [158, 177, 178], we

briefly shed light on the possibility of generalizing to additional transformations. First, one of the preprocessing steps for LSTM-based models, i.e., the tokenization of logic transformations, includes a special token <unk> designed for unknown transformations met in testing. Second, some out-of-vocabulary techniques in natural language processing (NLP) [179] can be adopted to improve the generalization capability to new transformations. Taking a step back, if the introduction of new transformations prompts significant modifications to the temporal models, either transfer learning tailored for NLP [180] or complete re-training can be potential solutions. These training efforts can be recognized as software updates, just as version updates in synthesis tools.

## 4.4 Conclusion

We propose a novel approach that aims to fulfill the two fundamental requirements of production-ready ML in EDA: high prediction accuracy and generalization capability. Our approach provides accurate and generalizable QoR estimations of logic synthesis flows by jointly considering the spatial information from circuit structures and the temporal information from synthesis flows. Two hybrid GNN-based models are developed accordingly: the first model uses a GNN to characterize circuit designs and includes a supernode to encode temporal information, while the second model comprises a GNN for spatial learning and an LSTM for temporal learning. Our evaluation results show that the testing MAPE on designs seen during training (i.e., transductive) and unseen during training (i.e., inductive) are no more than 1.2% and 3.1%, respectively. This demonstrates great generalization capability across designs, without the need for re-training.

# Chapter 5

# Reinforcement Learning for Fine-Grained Resource Allocation in High-Level Synthesis

In this chapter, we explore the potential of deep RL for fine-grained, flexible, and automated design space exploration in HLS, with the primary goal to provide either optimized solutions under user-specified constraints or Pareto trade-offs between different objectives (e.g., resource types and timing).

HLS streamlines the hardware design process by converting abstract behavioral descriptions into functionally equivalent RTL designs with varying resource and performance trade-offs. In addition to widely used commercial HLS tools for FPGA [151, 181] and ASIC [182], recent efforts have focused on improving HLS-based design quality through techniques such as loop transformation and memory allocation [183, 184], fast performance and/or resource prediction [118, 121], and design space exploration (DSE) [185]. Even though HLS tools have made significant progress, we observe several unresolved challenges as follows.

**Challenge 1: Concealed Optimization Opportunities**

The high-level abstraction of HLS programming styles, such as loops and function calls, conceals further optimization opportunities. While guidelines of HLS code optimization towards different design objectives are well investigated [186], they often focus on coarse-grained optimization in the loop/array/function-level and manual efforts for fine-grained exploration (such as in the operator-level) are still required.

**Challenge 2: Inflexible Design Exploration among Different Objectives**

With the increasing variety of workloads and the diverse performance, resource, and power targets, HLS designs typically require extensive DSE to satisfy design specifications. Existing DSE approaches in HLS usually sacrifice design latency for less resource or vise versa [185], leaving the flexible trade-offs among other objectives unexplored. Among several common academic HLS tools (e.g., LegUp [187], Dwarv [188], and Bambu [189]), only Bambu can generate trade-off implementations between latency and resource, but it still cannot balance between different types of resources; for commercial HLS tools, Vivado HLS [181] and Vitis HLS [151] do not provide Pareto solutions automatically, and Intel HLS [190] can only balance area-performance for memory systems. One unplumbed yet promising design exploration knob is to trade one type of resource for another (e.g., LUT and DSP in FPGA) while maintaining the latency unchanged, which currently can only be accomplished through laborious manual efforts (detailed examples in Chapter 5.1).

**IronMan: RL + GNN for Fine-Grained, Flexible, and Automated DSE**

To address these challenges, we propose an end-to-end framework, namely **IronMan**. The primary goal is to enable a fine-grained, flexible, and automated DSE to provide

either optimized solutions under user-specified constraints or Pareto trade-offs among different objectives (such as resource types and timing), which has not been achieved by existing HLS tools or DSE engines. IronMan consists of three components that seamlessly cooperate with each other, as illustrated in Figure 5.1. We briefly introduce these components and summarize our contributions as follows.

- **GPP**: we propose a highly accurate **G**NN-based **P**erformance **P**redictor for HLS designs. Notably, **GPP** predicts the *post-implementation metrics* rather than the *synthesized* results by HLS tools.

- **RLMD**: we propose a deep **RL**-based **M**ulti-objective **D**SE engine for optimal resource allocation strategies under user-specified constraints, which can also provide Pareto solutions between different objectives. In particular, RLMD is equipped with two different RL methods, offering the flexibility to choose a more proper optimization scheme for different cases.

- **CT**: we propose a **C**ode **T**ransformer, which *reveals concealed optimization opportunities* to achieve higher parallelism and shorter latency and allows for *flexible and fine-grained DSE*.

- **IronMan**: while each proposed component can independently contribute to the HLS community (performance prediction, DSE, and code transformation), we integrate them into a framework, IronMan, and demonstrate the end-to-end benefits on benchmarks from real-world applications.

- Experimental results show that, (1) GPP achieves high accuracy in predicting *post-implementation* performance, reducing the errors of HLS tools by 10.9× in resource utilization and 5.7× in critical path (CP) timing; (2) compared to meta-heuristic-based techniques, IronMan generates superior solutions reducing resource utiliza-

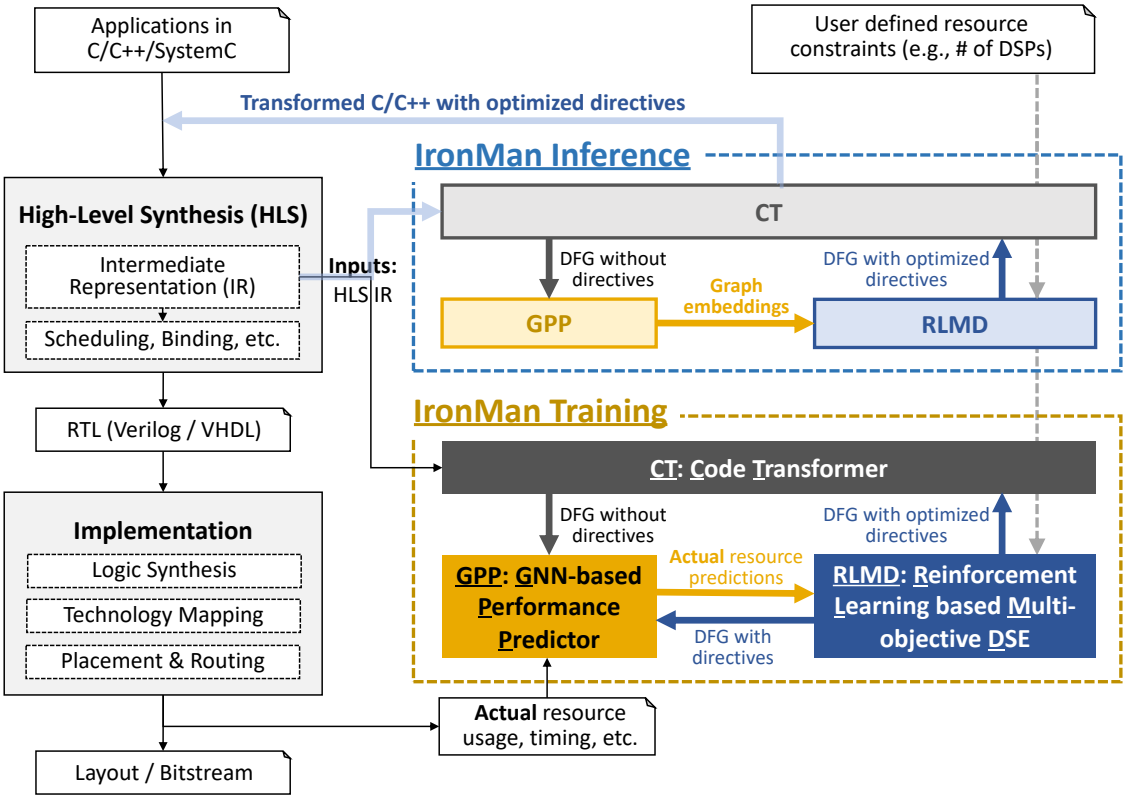


Figure 5.1: The proposed IronMan is a learning-based framework composed of CT, GPP, and RLMD. During training, IronMan takes HLS C/C++ code and IRs as inputs and the actual RTL performance (e.g., resource and timing) as the ground truth to train GPP and RLMD. During inference, the well-trained GPP provides graph embeddings and performance predictions to RLMD; the trained RLMD either finds optimized directives that satisfy user-specified design constraints such as available resources, or generates Pareto-solutions with various trade-offs between different resource types.

tion by 16.0% $\sim$ 29.5% and CP timing by 7.9% $\sim$ 16.5%; (3) under user-specified constraints, IronMan can find satisfactory solutions for over 96% of the cases, more than twice the number of cases handled by meta-heuristic-based techniques and with a significant speedup.

IronMan demonstrates the great potential of applying RL together with GNN in HLS, especially for the hard-to-solve problems such as timing estimation and optimization. IronMan is available at https://github.com/lydiawunan/IronMan.

This chapter is organized as follows: Chapter 5.1 summarizes the related work and explains the motivations; Chapter 5.2 introduces the overall framework of IronMan; Chapter 5.3 provides details of GPP and RLMD; Chapter 5.4 presents experimental results and discussions, followed by conclusions in Chapter 5.5.

## 5.1 Related Work and Motivation

Recently, there are surging research interests in applying ML-based techniques to improve HLS tools [8], covering two major aspects: fast and accurate performance prediction [23, 116, 117, 118, 121, 191], and efficient design exploration [185, 186, 192, 193, 194, 195]. The related work and motivation of exploiting GNNs for HLS performance predictions are detailed in Chapter 3.1. Here, we pay more attention to DSE in HLS.

### 5.1.1 Existing Approaches

Traditional DSE in HLS usually uses meta-heuristics, such as genetic algorithms (GA) [196, 197], simulated annealing (SA) [198], particle swarm optimization (PSO) [199], and ant colony optimization (ACO) [192]. In terms of ML-based DSE, several studies employ active learning to search Pareto solutions [124, 125, 200] or Bayesian optimization to explore design trade-offs [90, 201]. Some research efforts enhance meta-heuristics by incorporating ML algorithms. For example, Mahapatra et al. [194] use decision trees to improve the performance of SA; Wang and Schäfer [193] use ML to help decide the hyper-parameters of the meta-heuristics.

### 5.1.2 Why RLMD?

In spite of the success of existing DSE approaches, there are several limitations. First, classic meta-heuristics require explorations from scratch for every new design and do not benefit from previous experiences, which may result in long searching time and degraded solution quality. Second, many DSE approaches (either meta-heuristic-based or ML-based) need to invoke the synthesis and implementation process to validate the newly generated solutions during optimization, which could be time-consuming. Third, not all DSE approaches are suitable for large design spaces. For example, Bayesian optimization is effective only when the parameter space is small [162].

To overcome these limitations, we propose RLMD that adopts deep RL to learn optimal resource allocation strategies for three major reasons. First, the resource allocation problem in HLS is often formulated as a sequential decision-making problem, naturally falling into the realm of RL. Second, the relationship between resource allocation decisions and actual resource usage is not explicit, and the impact of one decision may not be immediately visible. RLMD can handle this delayed reward scenario by proactively exploring design knobs and learning policies through interactions with hardware performance metrics and user-specified constraints. Third, with the help of GPP, RLMD can make better use of past experiences, and once well-trained, it can run inference in seconds without invoking HLS tools or the implementation process, presenting excellent scalability even in exponentially increasing design spaces.

### 5.1.3 Why CT?

The high level abstractions in HLS can conceal fine-grained optimization opportunities or design trade-offs. Current DSE approaches mainly focus on the coarse-grained *loop/array/function-level* [186], rather than the fine-grained *operator-level*, which hinders

Table 5.1: Different approaches to meeting the DSP constraint (e.g., $\leq 3$) in a multiplication-accumulation function, leading to various clock cycles (latency), LUT usage, and CP timing. IronMan explores *CT + resource* approaches.

**Orig. Code: for (int i=0; i<8; i++) sum += a[i]*b[i];**

| | Method | Cycles | DSP | LUTs | CP (ns) |
|---|---|---|---|---|---|
| 1 | Original | 17 | 1 | 75 | 4.07 |
| 2 | unroll (factor=8, complete) | 2 | 8 | 100 | 5.04 |
| 3 | unroll (factor=4) | 4 | 4 | 87 | 4.83 |
| 4 | unroll (factor=3) | 8 | 3 | 109 | 7.44 |
| 5 | unroll + allocation (limit=3) * | 4 | 6 | 168 | 8.76 |
| 6 | Code Transform (CT) | 2 | 8 | 100 | 5.04 |
| 7 | CT + allocation (limit=2) | 5 | 3 | 196 | 9.91 |
| 8 | CT + allocation (limit=3) | 4 | 6 | 168 | 8.54 |
| 9 | ▷ CT + resource (5 Mul_LUT) | 2 | 2 | 1742 | 4.24 |
| 10 | ▷ CT + resource (4 Mul_LUT) | 2 | 2 | 1741 | 4.01 |
| 11 | ▷ CT + resource (3 Mul_LUT) | 2 | 3 | 1461 | 3.98 |

* HLS pragmas do not always behave as expected.

advanced optimization techniques for operator-level resource binding. Table 5.1 demonstrates a motivating example of fine-grained DSE. To explore the trade-offs between the DSP usage and the number of clock cycles (i.e., latency), the typical ways are to use *unroll* pragmas or manual loop-tiling, as in line 2-4. However, when the loop boundary (e.g., 8) is not divisible by the DSP constraint (e.g., 3), it results in a *partial unrolling* as line 4, introducing a undesired latency increment (from 4 to 8) and worsening the CP timing (from 5 ns to 7.4 ns). The nested loops further complicate this problem and make it much harder to balance between latency and resource (imagine a 5-layer nested loop with a DSP constraint of 17).

Motivated by the necessity of more flexible and fine-grained DSE, we propose CT, which breaks up the high-level abstractions by exposing operations in behavioral descriptions, fuses them into DFGs, and re-generates synthesizable C/C++ code with pragmas. CT enables the easy use of directives, such as *allocation* and *resource* pragmas, to conduct finer-grained DSE for resource and performance (Table 5.1 lines 7-11). Notably, IronMan explores the **CT + resource** approaches (lines 9-11) and achieves the best

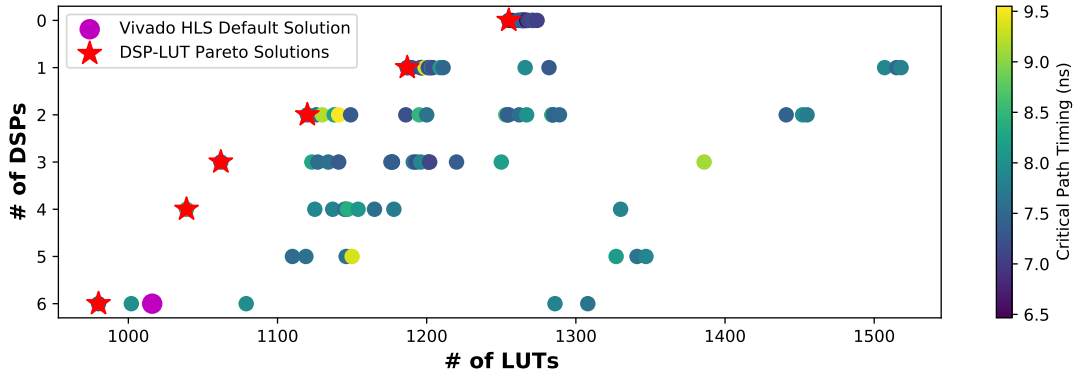


Figure 5.2: Pareto solutions between DSPs and LUTs on an FPGA. The default HLS solution is not on the Pareto frontier. It is non-trivial to obtain Pareto solutions in a large design space.

latency (i.e., 2) within the DSP constraint (i.e., $\leq 3$) without manual efforts.

### 5.1.4 Why IronMan?

Figure 5.2 elucidates the Pareto solutions between LUTs and DSPs, achieved by specifying certain multiplications computed by LUTs instead of DSPs. Apparently, the HLS default solution is not on the Pareto frontier, and finding Pareto solutions often requires exploring a large design space. Integrating **GPP**, **RLMD**, and **CT**, we propose the framework **IronMan**, either for optimized resource allocation under user-specified constraints or Pareto solutions between different resources. Our innovations stem from three aspects: (1) the fine-grained DSE enabled by CT that are not supported by any of the existing DSE approaches; (2) the efficient and scalable DSE for Pareto solutions, especially in extremely large design spaces; and (3) accurate performance predictions for both regular and irregular logics from early circuit design stages.

## 5.2 Overall Framework

Figure 5.1 shows the overall framework of IronMan, composed of three components: **CT**, **GPP**, and **RLMD**.

- **CT** is the interface between GPP/RLMD and HLS tools, which extracts DFGs after HLS front-end compilation to release more optimization opportunities and then re-generates synthesizable C/C++ code based on the optimized DFGs. Notably, CT respects pragmas set by users during the code transformation, and conducts complete loop unrolling and array partitioning only when there is no user-specified pragma. If the resource utilization achieved by RLMD is beyond the FPGA board availability, CT issues a warning, suggesting either relax the constraint or adjust the pragmas.

- **GPP** is a GNN-based performance predictor, which estimates *post-implementation* resource usage and timing of DFGs. GNNs [39, 139, 202] are adopted for three reasons: (1) DFGs are graphs, which are naturally suitable for GNNs to learn the underlying information from graph structures; (2) DFGs vary in topologies and sizes, and to generalize predictions to new or unseen graphs, it is necessary to use *inductive* GNNs [39] to learn fixed-size *graph embeddings*; (3) IronMan runs inference of trained GNN models during execution, which is orders of magnitude faster than running HLS tools.

- **RLMD** is an RL-based DSE engine. It takes the raw DFG, its graph embedding, and user-specified constraints as inputs, to optimize resource allocation strategies. RL is adopted for two main reasons: (1) the design space grows exponentially with the size of DFGs, different graph topologies, and various data precisions; an RL agent can explore design space proactively and learn from past experiences, and

a well-pretrained agent can generalize to new problems with minimal fine-tuning (FT) efforts; (2) by carefully defining reward functions, RL agents can achieve multi-objective optimization automatically, getting rid of manual efforts to craft useful heuristics. Since an informative and well-crafted state representation will significantly benefit the learning process in RL problems, we integrate GPP with RLMD, where the graph embeddings are naturally suitable for state representations in this problem. Consequently, the graph embeddings enable RLMD to better generalize across different DFG topologies, and GPP largely accelerates the training process of RLMD by quick evaluation of solutions generated by RLMD.

The inputs to IronMan are HLS C/C++ code and user-specified constraints. The outputs are re-generated code with optimized HLS directives, either meeting user-specified constraints (e.g., resource constraints) or providing Pareto solutions between different optimization objectives. The entire flow has three major steps.

- **Step 1.** CT takes in the C/C++ code written by designers and extracts the corresponding DFG after HLS front-end compilation to release more optimization opportunities.

- **Step 2.** The extracted DFG is fed into GPP and RLMD to find optimized resource allocation solutions by assigning multiplications to DSPs or LUTs.

- **Step 3.** After RLMD completes the assignment of resource pragmas, the optimized DFG is converted back to functionally equivalent C/C++ code with properly assigned pragmas by CT. An example of the regenerated C++ code is shown in Figure 5.3(a).

As a case study of IronMan, the specific problem solved is to find a resource allocation solution that strictly meets the DSP constraint, or to find Pareto solutions between

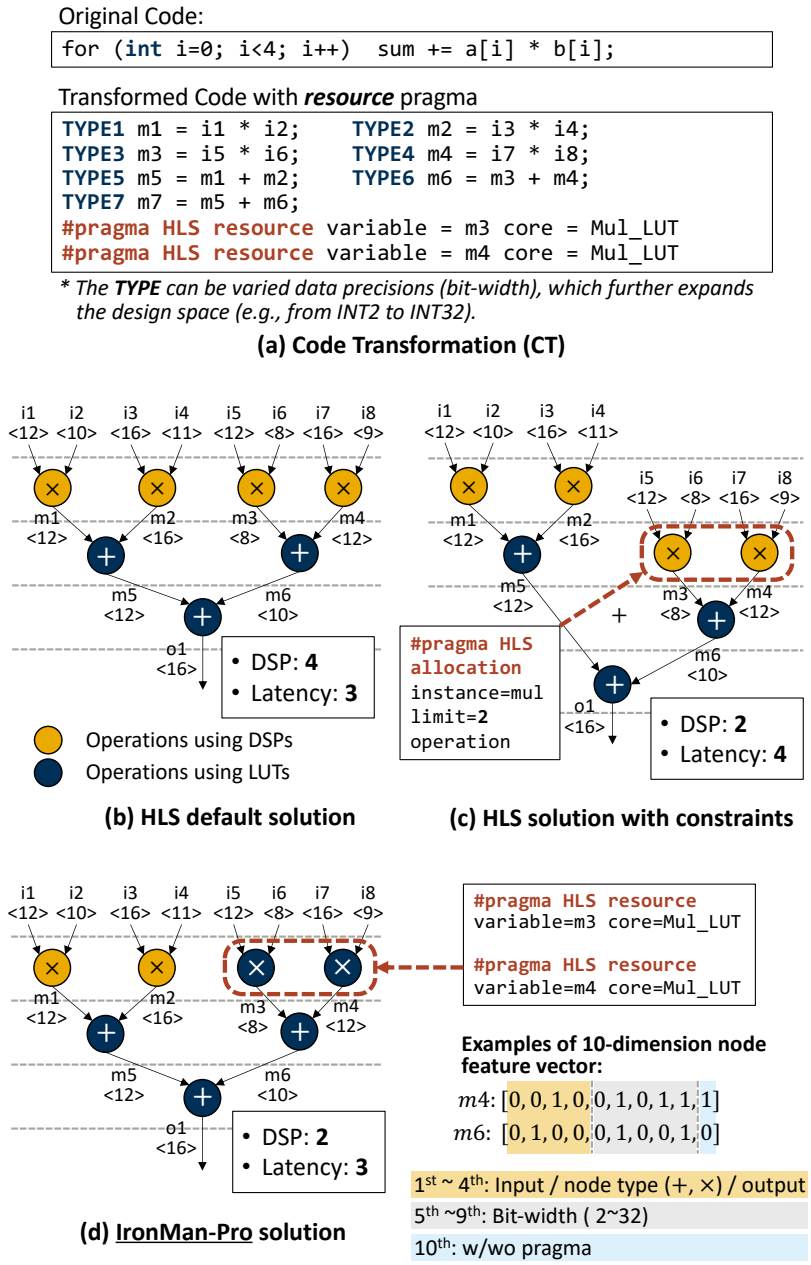


Figure 5.3: Example of IronMan solution. (a) Original HLS code and transformed code with resource pragma, indicating the importance of CT for IronMan solutions; (b) HLS default solution with four DSPs and a latency of 3, note that each intermediate operator may have various bit-width, e.g., ⟨12⟩ means a 12-bit data precision; (c) HLS solution with naive constraints, using two DSPs while increasing latency from 3 to 4; (d) IronMan solution, with two DSPs and an unchanged latency of 3.

DSPs and LUTs (or CP timing) on FPGAs, without sacrificing the computation latency. For simplicity, the DFGs only have additions and multiplications, where RLMD decides whether to assign the directive *#pramga HLS resource core=Mul_LUT* to each multiplication operation, to minimize LUTs within DSP constraints. The user-specified constraints is the number of DSPs that can be used in a design. For FPGA designs, mapping multiplications on LUTs is a common practice [203]. From the resource aspect, DSPs are often scarce resources, whereas LUT resources are usually more abundant. From the execution efficiency aspect, DSPs could be slower than LUTs. For example, the delay of a multiplier mapped on DSP blocks is around 4-5 ns, while the delay of a LUT-based implementation is around 0.7 ns [204, 205].

Figure 5.3 exemplifies a solution provided by IronMan given the constraint of using two DSPs. Figure 5.3(b) is the default HLS solution with four DSPs and a latency of 3; Figure 5.3(c) is a naive solution using *#pragma HLS allocation instance=mul limit=2* to enforce two DSPs, resulting in an increased latency from 3 to 4; Figure 5.3(d) shows the solution of IronMan with two DSPs and an unchanged latency of 3, using *#pragma HLS resource variable=⟨var⟩ core=Mul_LUT*.

## 5.3 Proposed GPP and RLMD

Since GPP provides inputs and performance predictions for RLMD, we first introduce the structure of GPP, and then discuss the detailed formulation of RLMD.

### 5.3.1 GPP: GNN-based Performance Predictor

The key role of a GNN is to extract adequate information about node types, graph topology, and connectivity within a large DFG, and to encode the information into low-dimensional vector representations that can be used for either downstream tasks or high-

accuracy performance prediction.

### Node Feature Vector

In a DFG, each node is encoded into a 10-dimension node feature vector, as the example shown in Figure 5.3(d). The 1st-4th dimension use one-hot representations to encode the node types, including input nodes, intermediate nodes/operations (i.e., additions and multiplications), and output nodes. The 5th-9th dimension encode the data precision of a node, which in this work ranges from INT2 to INT32. We use a binary representation to encode the precision minus one, so the bit-width can be expressed in 5 bits. The 10th dimension indicates whether an HLS directive *#pramga HLS resource* is applied to this node. Note that such an encoding scheme can be easily extended to support more types of nodes/operations or pragmas.

### Graph Embedding

We employ the GCN [139] to predict the performance of synthesized circuits. Formally, the layer-wise propagation of each graph convolutional layer can be formulated as follows:

$$H^{(l+1)} = \sigma(\widetilde{D}^{-\frac{1}{2}}\widetilde{A}\widetilde{D}^{-\frac{1}{2}}H^{(l)}W^{(l)}), \tag{5.1}$$

where $\widetilde{A} = A + I_N$ is the sum of the adjacency matrix $A$ of the DFG $\mathcal{G}$ and the identity matrix $I_N$. The addition of the identity matrix means adding a self-loop on each node, so that the updated node embedding will include the impact from both the neighbors and itself. $\widetilde{D}$ is a diagonal matrix to normalize the adjusted adjacency matrix $\widetilde{A}$, where $\widetilde{D}_{ii} = \sum_j \widetilde{A}_{ij}$. $W^{(l)}$ is a layer-specific trainable weight matrix, and all the nodes within the $l^{th}$ layer share the same weights $W^{(l)}$. $\sigma(\cdot)$ is the activation function. $H^{(l)} \in \mathbb{R}^{n \times d}$ is the matrix of activations from the $l^{th}$ layer, which stacks the hidden vectors of each node,

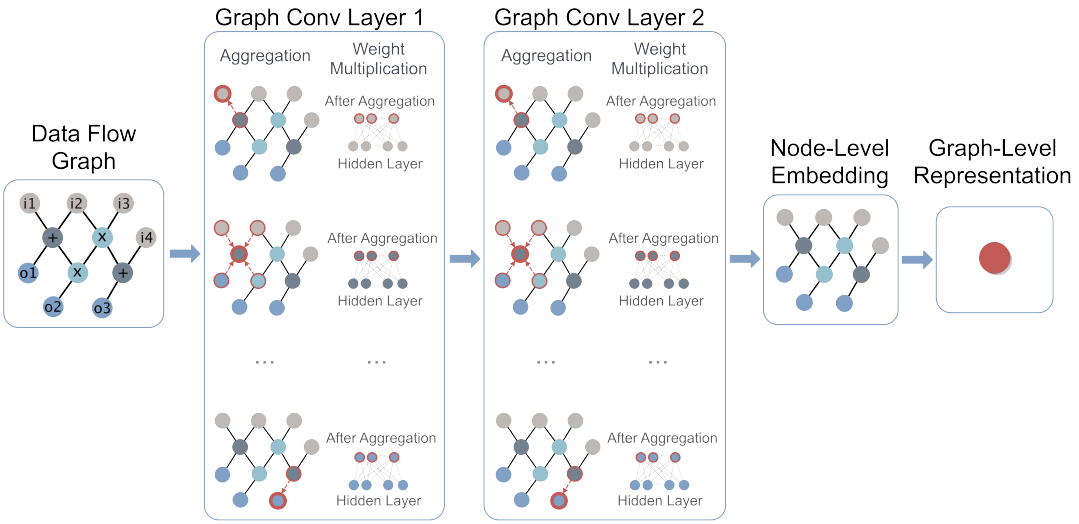


Figure 5.4: Example of employing two graph convolutional layers to generate graph representations.

assuming that there are $n$ nodes in total with feature dimension $d$. $H^{(0)}$ is the matrix stacking all the input node feature vectors.

In our proposed structure of GPP, the inputs to the first graph convolutional layer are adjacency matrices and node feature matrices of DFGs. In each graph convolutional layer, the node embedding is updated by aggregating feature vectors from its neighbors and multiplying with the corresponding weight matrix. One node can receive information from multi-hop neighborhoods by stacking multiple layers. Figure 5.4 illustrates an example structure that employs two graph convolutional layers to process DFGs. After multiple layers, the learned node embeddings are summarized by a mean pooling to create a graph representation.

This graph representation vector is then passed to a feed-forward network with three FC layers and leaky ReLU ($\alpha = 0.1$) activations to generate a graph embedding. The last layer is the output of the GNN, including a single unit with the ReLU activation to provide prediction results of resource usage and timing. In order to separately predict LUT/DSP utilization and CP timing, we use three GNN models of the same structure,

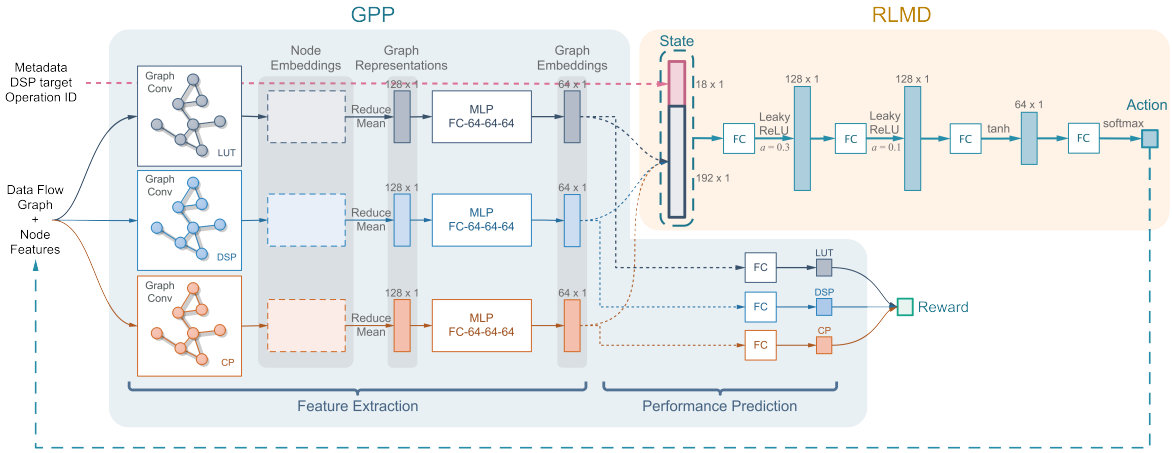


Figure 5.5: The structure of GPP and RLMD. GPP encodes information of DFG adjacency matrices and node features, to make predictions of LUT/DSP/CP. Concatenating the graph embeddings provided by GPP with the metadata of the input DFG, RLMD then outputs a binary probability distribution $\pi(a_t|s_t)$ of whether to use LUTs for multiplication computation on the current node. For the actor-critic method, RLMD also outputs a scalar as the state-value function.

as illustrated in the left part of Figure 5.5.

### Integration with RLMD

To integrate with RLMD, we concatenate the three graph embedding vectors, which focus on different characteristics of DFGs, with metadata of the input DFG, to create an input vector for RLMD. The DFG metadata include the size of the DFG (i.e., the number of input/intermediate/output nodes and the number of edges) and the number of multiplications in this DFG. Given predictions of LUT/DSP/CP, solutions generated by RLMD can be quickly evaluated, providing feedback to further improve the policy of RLMD.

### 5.3.2 RLMD: Reinforcement Learning based Multi-objective Design Space Exploration

We describe the problem formulation, the employed RL algorithms, and the training and fine-tuning procedures of RLMD.

#### RL Formulation

The resource allocation problem in HLS, as a typical RL [48] problem, can be formulated as a Markov Decision Process (MDP), with four key components.

- States: the set of possible states. In this problem, a state can be every possible partially assigned DFG.
- Actions: the set of eligible actions under a state. In this problem, given the current state and the currently considered node of the DFG, the available action is whether to assign a certain directive to this node.
- State transition: given a state and an action, the probability distribution of next states.
- Reward: the immediate reward of taking an action in a state. In this problem, the reward is 0 for all intermediate actions, with an exception for the last action where the reward is the evaluation of the fully assigned DFG subject to user-specified constraints.

Specifically, the state at time step $t$ is defined as $s_t$, which is a concatenation of features including the ID of current node to assign a directive, metadata of the DFG, the DSP constraint (either user-specified or automatically generated for Pareto solution exploration), and a $192 \times 1$ graph embedding vector that describes the current status

of the DFG. The action $a_t$ is a valid assignment of a directive to the $t$-th node, i.e., whether to use LUTs for multiplication computation on this node. We define the reward $r_t$ as a negative weighted sum of predicted LUTs, CP timing, and the difference between predicted and target DSPs, as follows:

$$r_t = \begin{cases} -\alpha \mathrm{LUT}_p - \beta |\mathrm{DSP}_t - \mathrm{DSP}_p| - \lambda \mathrm{CP}_p, & t = T \\ 0, & 0 < t < T \end{cases}. \tag{5.2}$$

where $\alpha$, $\beta$ and $\lambda$ are hyper-parameters; $\mathrm{DSP}_t$ is the target number of DSPs; $\mathrm{DSP}_p$, $\mathrm{LUT}_p$, and $\mathrm{CP}_p$ are the predicted values by GPP; $T$ is the total number of time steps.

At the initial state $s_0$, all the multiplication nodes in a DFG are unassigned. At each time step $t$, the RL agent observes the current state $s_t$, takes an action $a_t$, receives a reward $r_{t+1}$, and arrives at a new state $s_{t+1}$. The nodes are assigned with directives sequentially based on their node IDs. Given $T$ multiplication nodes in total, the final state $s_T$ corresponds to a DFG completely assigned with proper directives. The goal is to maximize the expected rewards received. The entire process of resource allocation in RLMD is shown in Figure 5.6.

### RLMD Training

RLMD is equipped with two different RL methods, providing the flexibility to choose a more proper optimization scheme for different cases. The entire training procedure is summarized in Algorithm 1.

**Actor-critic (AC) method.** We adopt the actor-critic method with Monte-Carlo learning [48]: the actor aims to learn an optimal policy $\pi_\theta(a_t|s_t)$ parameterized by $\theta$, which is a probability distribution of valid actions under the current state; the critic approximates the state-value function $V(s_t) = \mathbb{E}_\pi[\sum_{k=0}^{T-t-1} \gamma^k r_{t+k+1}|s_t]$ by parameters $w$,

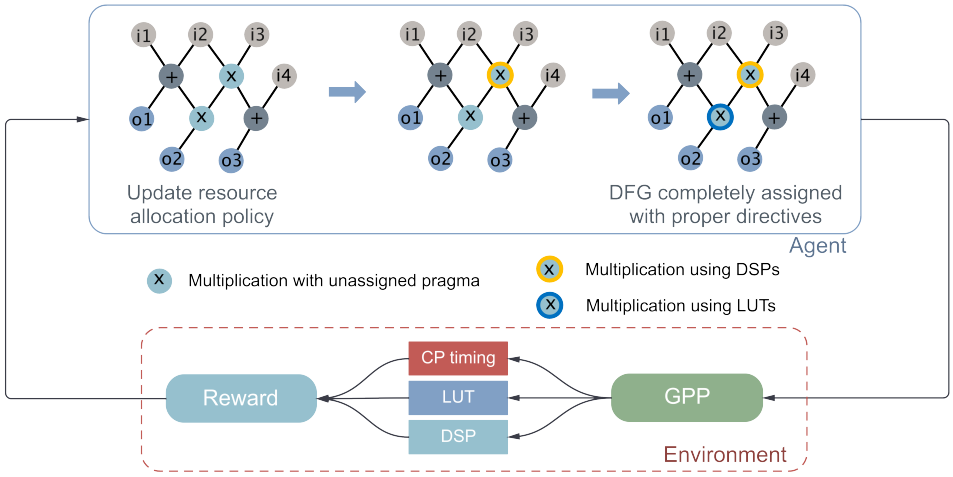


Figure 5.6: Overview of the resource allocation process in RLMD. Given a DFG, RLMD sequentially decides whether to assign resource pragmas for every multiplication. After the DFG is completely assigned with resource allocation pragmas, GPP quickly evaluates this solution, and the reward is computed accordingly to improve the resource allocation strategy.

which is an estimate of accumulated rewards starting from state $s_t$ to $s_T$ following policy $\pi$, measuring the goodness of this state. The $\gamma \in (0, 1]$ is the discount factor. By Monte-Carlo learning, the parameters are updated only after one complete episode (i.e., one complete assignment process of a DFG), leading to following updates:

$$\delta_i = \gamma^{T-i-1} r_T - V_w(s_i), \tag{5.3}$$

$$\Delta w \propto \sum_{i=0}^{T-1} \delta_i \nabla_w V_w(s_i), \tag{5.4}$$

$$\Delta \theta \propto \sum_{i=0}^{T-1} \delta_i \nabla_\theta \log \pi_\theta(a_i|s_i), \tag{5.5}$$

where $T$ is the total time steps in one episode. Through repeated episodes (i.e., sequences of states, actions, and rewards), the actor learns optimized policy that will maximize cumulative rewards.

**Algorithm 1:** AC/PG for resource allocation optimization.

Generate tuples in the form of $[DFG_{index}, DSP_t]$ and duplicate each tuple for $m$ times;
Put all tuples together and shuffle the order;
(a) ***For AC***: initialize the actor $\pi_\theta(a|s)$ and the critic $V_w(s)$ with weights $\theta$ and $w$;
(b) ***For PG***: initialize the policy network $\pi_\theta(a|s)$;
Initialize the episode counter $i \leftarrow 0$;
**while** $i < episode_{max}$ **do**
  $t \leftarrow 0$;
  Get the $DFG_{index}$ and $DSP_t$;
  Initialize state $s_0$ based on $DFG_{index}$ with all the multiplication nodes unassigned of directives;
  **while** $t < T$ **do**
    Compute and store $\pi_\theta(a_t|s_t)$ and $V_w(s_t)$;
    Take action $a_t$ based on $\pi_\theta(a_t|s_t)$ with $\epsilon$-greedy algorithm;
    Receive the reward $r_{t+1}$ as defined in Equation (5.2);
    Get $s_{t+1}$ from the updated DFG;
    $t \leftarrow t + 1$;
  **end while**
  (a) ***For AC***: update $\theta$ and $w$ according to Equations (5.3)-(5.5);
  (b) ***For PG***: update $\theta$ according to Equations (5.6)-(5.7);
  $i \leftarrow i + 1$;
**end while**

**Policy-gradient (PG) method.** Instead of the setting with an "actor" and a "critic" as in AC, the purely policy-based methods only have one actor network, i.e., the policy network. As aforementioned, in AC-based methods, the critic networks are trained to approximate the underlying value functions of states, which provide guidance to train the policy networks. One thing worth noting is that when the effective rewards are too sparse and irregular, the critic network may not learn well and thus impose adverse impacts on the learning of the policy networks [206].

Taking this into consideration, we also explore a policy-gradient method, REINFORCE [207], in an attempt to expand the choices available in the toolbox of IronMan. In this case, we only need to learn an optimal policy $\pi_\theta(a_t|s_t)$ parameterized by $\theta$. Similarly, the parameters in the policy network are updated after the completion of

one episode, as follows.

$$G_i = \gamma^{T-i-1} r_T, \tag{5.6}$$

$$\Delta\theta \propto \sum_{i=0}^{T-1} G_i \nabla_\theta \log \pi_\theta(a_i|s_i), \tag{5.7}$$

where $G_i$ is the discounted reward at time step $i$, and $T$ is the total time steps in one episode.

**Generalization across DFGs.** Our ultimate goal is to enable RLMD to generate higher-quality results and transfer knowledge across various DFGs as it gains experience from exploring resource allocation strategies on more and more DFGs. Thus, we formally formulate the overall optimization objective function as

$$\mathcal{J}(\theta, w, G) = \frac{1}{K} \sum_{g \in G} \mathbb{E}_{g,p\sim\pi_\theta}[R_{g,p}], \tag{5.8}$$

where $\mathcal{J}(\theta, w, G)$ measures the expected cumulative rewards over all training DFGs. The DFG dataset $G$ has $K$ different DFGs, each of which is denoted as $g$. $R_{g,p}$ is the episode reward (i.e., $r_T$ in Equation (5.2)) under the resource allocation solution $p$ on the DFG $g$. To get better exploration during training, we apply $\epsilon$-greedy algorithm for action selections [48].

### RLMD Fine-tuning

Given a new DFG, the simplest way is to directly apply the pre-trained RLMD for inference, which can generate a solution within a second. When higher quality solutions are expected, the pre-trained RLMD can be further fine-tuned on a particular DFG. The fine-tuning step provides the flexibility to balance between a quick solution using the pre-trained RLMD (which has learned rich knowledge of resource allocation strategies on

other DFGs) and a longer yet better one for a particular DFG.

### 5.3.3 Multi-Objective Optimization

To enable flexible and multi-objective optimization, we define the multi-objective optimization function as follows:

$$\mathcal{U}(\mathrm{LUT}, \mathrm{CP}|\mathrm{DSP}_{target}) = \mu\frac{\mathrm{LUT}}{q} + (1-\mu)\mathrm{CP}, \tag{5.9}$$

where $\mu$ indicates the relative attention paid on different metrics, and $q$ is a constant to balance the numerical scales of LUT usage and the CP timing. The goal is to minimize $\mathcal{U}$, i.e., to minimize the LUT usage and CP timing simultaneously, given the DSP constraint. The larger $\mu$ is, the more importance is given on the LUT usage during optimization, and vice versa.

As defined in Equation (5.2), the reward function is a negative weighted sum of multiple metrics, which automatically enables multi-objective optimization. By adjusting the weights (i.e., hyper-parameters) of different metrics of interest, RLMD can figure out various trade-offs while complying with user-specified constraints. To relate the optimization objective function with the reward function, we let $\alpha = \frac{\mu}{q}$ and $\lambda = 1-\mu$. If more metrics of interest or another form of multi-objective optimization function would be desired, the reward function can be crafted accordingly.

## 5.4 Experiment

We present the experiment setup and evaluation of GPP, RLMD, as well as the end-to-end framework IronMan.

### 5.4.1 Experiment Setup

For GPP and RLMD training, we build a dataset containing both synthetic and real-world DFGs. For synthetic DFGs, we randomly generate 47 different topologies, each of which has 100 to 200 operations (i.e., intermediate nodes) of either multiplication or addition. On top of each distinct topology, we further generate 100 sets of directives, which specify a subset of multiplications to be implemented by LUTs rather than DSPs. This makes up 4,700 (i.e. 47 $\times$ 100) different synthetic DFGs. For real-world DFGs, we consider eight applications from MachSuite [135], CHStone [136] and PolyBench/C [137]: *gemm, kernel_2mm, kernel_durbin* (small, large), *spmv, stencil3d* (small, large), and *kernel_adi*. Similarly, we randomly generate 100 sets of directives per application, making up 800 different DFGs for real cases. The ground-truth (actual) resource usage (LUT/DSP) and CP timing are synthesized by Vitis HLS [151] and implemented by Vivado‘[152] targeting Xilinx Ultra96 part xc7z020clg484. The target frequency is 100 MHz.

#### Training Process

To demonstrate the generalization capability of IronMan across different DFGs and applications, GPP and RLMD are trained on part of DFGs from the dataset and evaluated on rest of them. The training set consists of 42 different topologies and 4 real-case applications (*kernel_durbin* and *stencil3d*), involving 4,600 DFGs in total.

GPP is trained via regression to minimize the mean-squared logarithmic errors for DSPs and CP timing, and the mean absolute errors for LUTs, respectively. In terms of hyper-parameter selection, GPP is trained over 200 epochs with a batch size of 32; the Adam optimizer is applied with an initial learning rate of 0.01 decaying exponentially.

After GPP is well-trained, it is integrated with RLMD to help with RLMD training. To train RLMD, we provide tuples in the form of $[\mathrm{DFG}_{index}, \mathrm{DSP}_{target}]$ to the RL agent,

and the optimization goal is to maximize the average cumulative rewards on all the tuples, so that the agent can learn resource allocation strategies under different DSP constraints and across different DFGs. There are 1,125 different tuples in total, and each tuple appears eight times during the training process, amounting to 9,000 episodes. Once RLMD is trained, it can be directly applied on new DFGs and generate solutions by inference in seconds. To make the trained RLMD specialize for a new DFG, we conduct fine-tuning with additional 500 episodes of training, after which better solutions will be generated at the cost of longer runtime. The parameters in RLMD are learned by the Adam optimizer with the learning rate of 0.008. We empirically set the discount rate $\gamma = 0.95$, and the exploration rate $\epsilon = 0.08$ decaying exponentially. In the reward function, we have $\beta = 5$ and $q = 500$; three different scenarios to trade-off between LUT usage and CP timing are considered, where $\mu$ is set as 0.1, 0.5, or 0.9.

As for the implementation of IronMan, GPP is implemented with StellarGraph and RLMD is implemented in TensorFlow 2. Experiments are performed on a Linux host with a 64-core Intel Xeon Gold 5218 CPU (2.30 GHz) and Nvidia RTX 2080Ti GPUs.

### 5.4.2 Evaluation

IronMan is compared with meta-heuristic-based approaches, which are widely applicable to many HLS DSE problems [185]. Meta-heuristics based DSE techniques are often nature-inspired, several representatives include SA [15], GA [208], PSO [199], and ACO [18]. Specifically, SA is a pseudo-random optimization approach; GA is an evolutionary approach [209]; PSO and ACO make use of swarm intelligence [210].

To demonstrate the capability of IronMan, in the evaluation of GPP, we compare with the commercial tool Vitis HLS [151] and the ML-based circuit performance predictor Pyramid [121]; in the evaluation of RLMD, whose toolbox includes AC and PG with or

without fine-tuning, we compare against SA, GA, PSO, and ACO.

### GPP Evaluation

Since stacking many GCN layers may bring the concern of over-smoothing [211], we explore the structure of GPP regarding different numbers of GCN layers adopted for feature extraction. As shown in Figure 5.7, the structure that employs two GCN layers generally outperforms others, which is considered as the final structure of GPP.

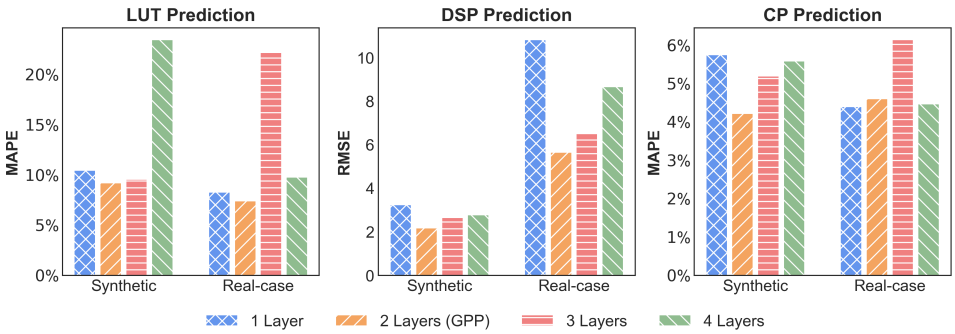


Figure 5.7: Comparison of applying different numbers of GCN layers, where the prediction accuracy of LUT and CP timing is measured by MAPEs and that of DSP is measured by RMSEs.

GPP is evaluated on both synthetic and real-case DFGs. Figure 5.8 compares GPP predictions with HLS synthesis reports regarding LUT, DSP, and CP timing. For LUT utilization, the mean absolute percentage errors (MAPEs) of GPP on synthetic and real-case DFGs are 7.4% and 9.2%, respectively, whereas the MAPEs of Vitis HLS are 122.4% and 92.2%, respectively. For DSP utilization, the prediction accuracy is measured by root-mean-square error (RMSE) since MAPE is not applicable when the ground truth appears to be zero. GPP achieves 5.6 and 2.1 in RMSE for synthetic and real-case DFGs, while Vitis HLS reaches 26.9 and 19.7, respectively. For CP timing, the MAPEs of GPP are 4.2% and 4.6% on synthetic and real-case DFGs, whereas the MAPEs of Vitis HLS are 7.7% and 42.1%. On average, GPP reduces the prediction error of Vitis HLS by

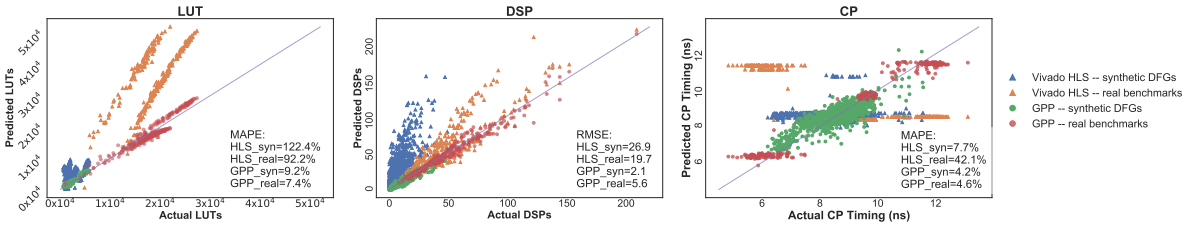


Figure 5.8: GPP predictions on resource utilization (LUTs and DSPs), and CP timing. 10.9× in resource utilization and 5.7× in timing.

Pyramid [121] is also an ML-based framework for resource and timing prediction. The major difference between GPP and Pyramid is the features required for predictions. Pyramid needs 72 features from HLS reports as inputs, which enforces the running of HLS to get VHDL designs, possibly consuming hours for large designs; whereas GPP can make high-accuracy predictions simply from raw DFGs (within a second). Pyramid considers four ML models and an ensemble of these four, none of which includes graphical structures. The reported results show that the averaged prediction error of a single ML model is 17.8% for resource and 17.3% for timing, with the ensemble reaching 5.5% for resource and 4.1% for timing.

### RLMD and IronMan Evaluation

We evaluate IronMan in terms of both Pareto solutions and the solutions found under user-specified constraints. All the generated solutions are synthesized by Vitis HLS [151] and implemented by Vivado [152], indicating that the reported resource usage (LUT/DSP) and CP timing are post-implementation.

**Pareto solutions.** Regarding the Pareto solutions between LUTs or CP timing and DSPs, Figure 5.9 and Figure 5.10 compare RLMD with GA, SA, PSO, and ACO with respect to synthetic cases and real-case applications, respectively. Obviously, RLMD, either with AC or PG method, outperforms GA, SA, PSO, and ACO by a large margin. In terms of multi-objective optimization, given DSP usage constraints, the solutions found

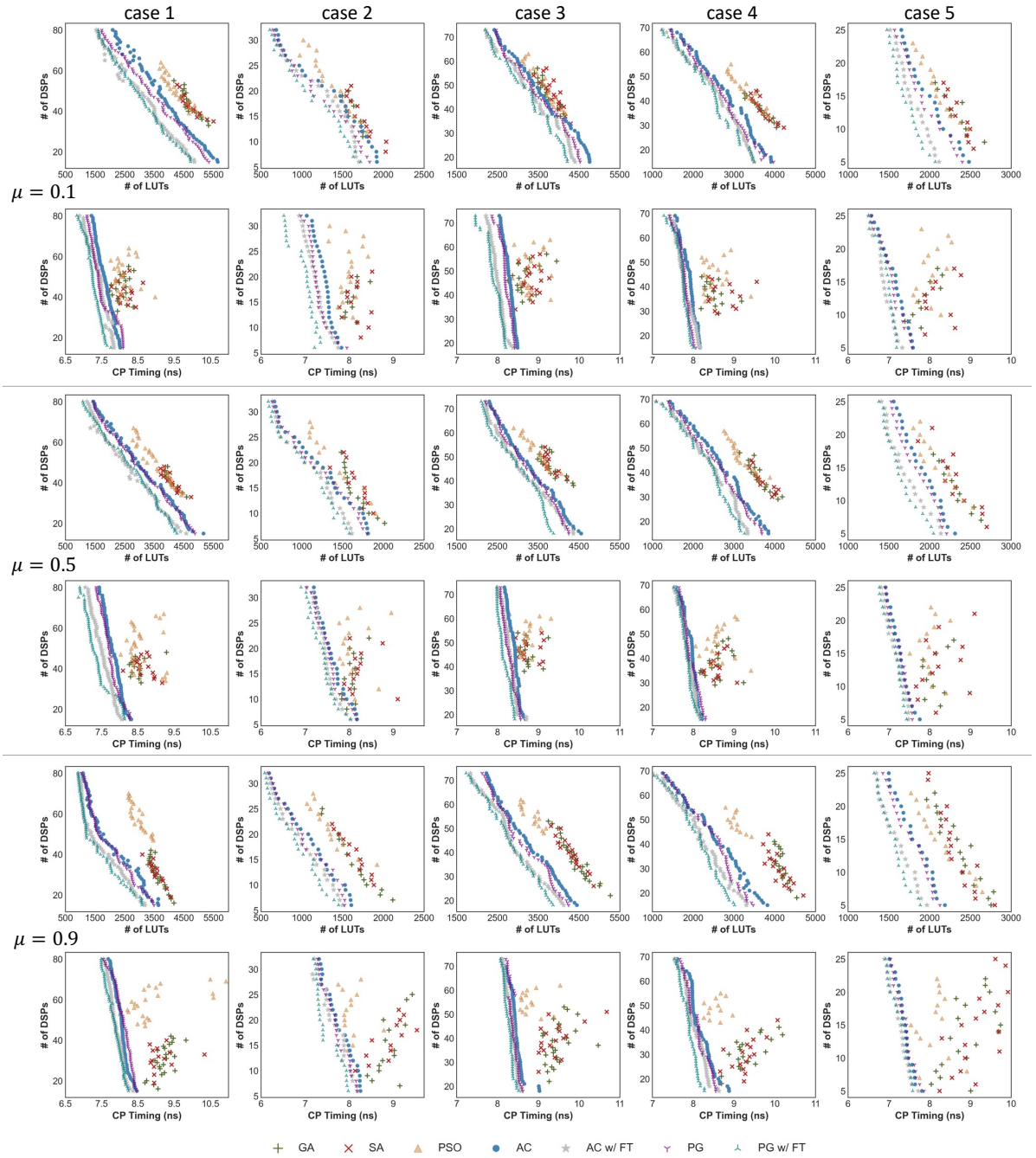


Figure 5.9: Pareto solutions found by RLMD, SA, GA, and PSO on five synthetic cases, with unchanged latency (i.e., the number of clock cycles of the synthesized design). The toolbox of RLMD involves AC, PG, either with or without a fine-tuning step. Different settings of $\mu$ indicate that different importance is assigned to LUT utilization and CP timing during the optimization.

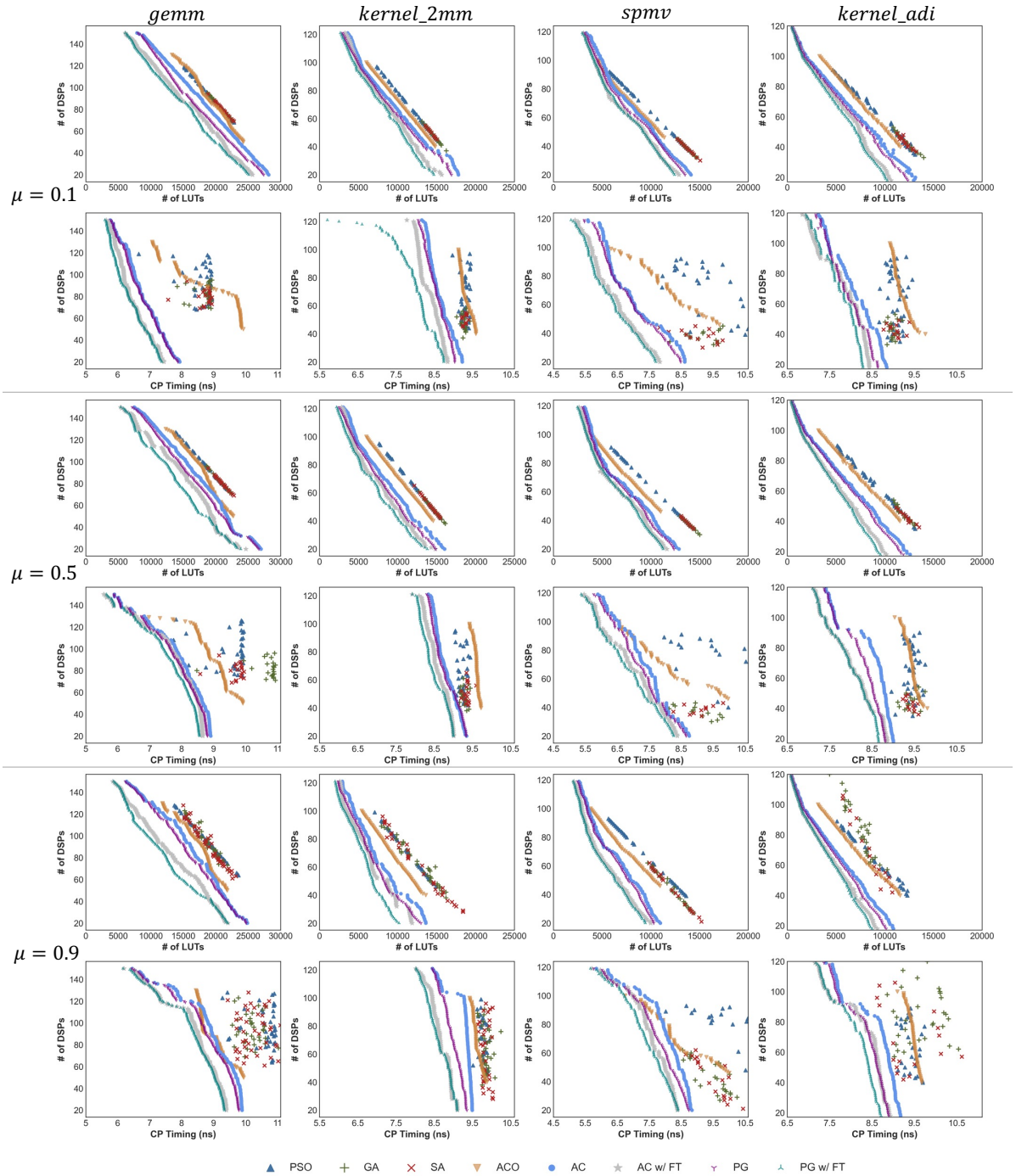


Figure 5.10: Pareto solutions found by RLMD, SA, GA, PSO and ACO on four real–case benchmarks, *gemm*, *kernel_2mm*, *spmv*, and *kernel_adi*, with unchanged latency (i.e., the number of clock cycles of the synthesized design). The toolbox of RLMD involves AC, PG, either with or without a fine-tuning step. Different settings of $\mu$ indicate that different importance is assigned to LUT utilization and CP timing during the optimization.

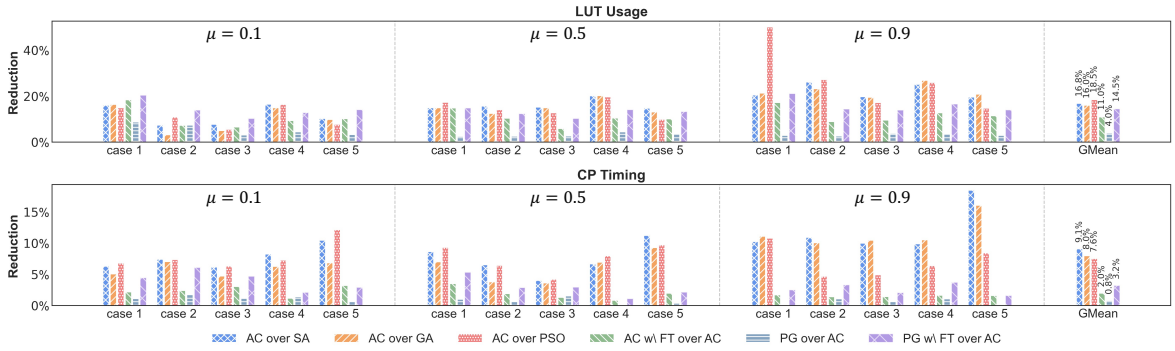


Figure 5.11: Statistics of reduction in LUT utilization and CP timing given the same number of DSPs, comparing RLMD with SA, GA, and PSO on five synthetic cases.

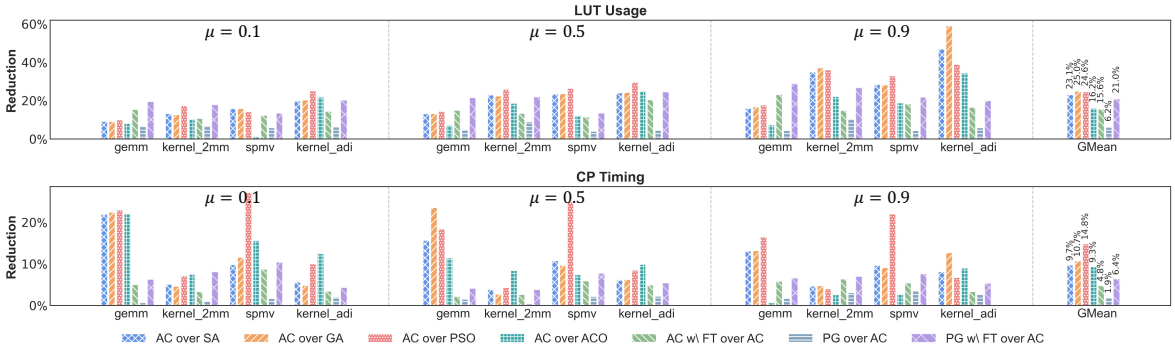


Figure 5.12: Statistics of reduction in LUT utilization and CP timing given the same number of DSPs, comparing RLMD with SA, GA, PSO, and ACO on four real-case benchmarks: *gemm*, *kernel_2mm*, *spmv*, and *kernel_adi*.

with $\mu = 0.9$ often consume fewer LUTs but larger (worse) CP timing, compared with those found with $\mu = 0.1$. This indicates that RLMD can properly balance between LUT usage and CP timing when different importance is assigned to different metrics, whereas the heuristic-based methods cannot explicitly leverage the trade-offs among multiple objectives.

For synthetic cases, Figure 5.11 depicts the statistics of solutions presented in Figure 5.9. When applying AC, given the same number of DSPs, solutions found by RLMD outperform those found by SA, GA, and PSO, attaining a decrease of 16.8%, 16.0% and 18.5% in terms of LUT utilization, as well as a decrease of 9.1%, 8.0% and 7.6% in terms of CP timing. In general, a slightly better set of solutions can be found by RLMD when employing PG, which consumes 4.0% fewer LUTs and 0.8% shorter CP timing,

compared with applying AC. After fine-tuning (FT), both AC and PG achieve additional reduction in LUT utilization and CP timing. For brevity, the fine-tuned solutions are contrasted against those found by AC. The LUT utilization is further decreased by 11.2% and 14.5% when using AC with FT and PG with FT, respectively. For CP timing, additional reduction of 2.0% and 3.2% are obtained by AC with FT and PG with FT, respectively.

Similarly, Figure 5.12 displays the statistics of the solutions of real-case applications from Figure 5.10. Given the same number of DSPs, RLMD that applies AC can find solutions surpassing SA, GA, PSO, and ACO, with a decrease of 23.1%, 25.0%, 24.6% and 16.2% in terms of LUT utilization, and with a decrease of 9.7%, 10.7%, 14.8% and 9.3% in terms of CP timing; RLMD that applies PG demonstrates better solutions compared with applying AC, which manages to use 6.2% fewer LUTs and 1.9% shorter CP timing. More significant improvement is displayed after performing FT for both AC and PG. The LUT utilization is further decreased by 15.6% and 21.0% when using AC with FT and PG with FT, respectively. As for CP timing, additional reductions of 4.8% and 6.4% are obtained by AC with FT and PG with FT, respectively. We also measure the quality of different approaches by using the average distance from reference set (ADRS) [212], as shown in Table 5.2.

These promising results show great potentials of applying RL for DSE in HLS. Through trials and interactions with GPP and user-specified constraints, RLMD is able to gradually understand which directive should be assigned to which node, and proactively learn proper resource allocation strategies by balanced exploration and exploitation. By contrast, one underlying assumption in GA is that the offspring of two strong individuals among a population is often stronger, which is not the case in DSE for HLS problems, thus reducing its effectiveness. Likewise, PSO is also a population-based stochastic optimization technique, and the key difference between PSO and GA is that PSO does

Table 5.2: ADRS of Pareto solutions found by RLMD, SA, GA, PSO, and ACO on 4 real-case benchmarks. The toolbox of RLMD involves AC, PG, either with or without a fine-tuning step.

| **LUT** | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **Design** | **SA** | **GA** | **PSO** | **ACO** | **AC** | **AC-FT** | **PG** | **PG-FT** |
| ***gemm*** | 0.429 | 0.441 | 0.351 | 0.23 | 0.153 | 0.039 | 0.121 | 0.001 |
| ***kernel_2mm*** | 0.613 | 0.608 | 0.36 | 0.235 | 0.147 | 0.067 | 0.086 | 0 |
| ***spmv*** | 0.718 | 0.709 | 0.251 | 0.151 | 0.104 | 0.018 | 0.075 | 0.001 |
| ***kernel_adi*** | 1.406 | 1.418 | 0.794 | 0.473 | 0.112 | 0.038 | 0.086 | 0.001 |
| **Mean** | 0.792 | 0.794 | 0.439 | 0.272 | 0.129 | 0.041 | 0.092 | 0.001 |
| **CP** | | | | | | | | |
| **Design** | **SA** | **GA** | **PSO** | **ACO** | **AC** | **AC-FT** | **PG** | **PG-FT** |
| ***gemm*** | 0.223 | 0.237 | 0.209 | 0.132 | 0.05 | 0.012 | 0.039 | 0 |
| ***kernel_2mm*** | 0.13 | 0.123 | 0.108 | 0.101 | 0.063 | 0.022 | 0.048 | 0 |
| ***spmv*** | 0.169 | 0.173 | 0.162 | 0.12 | 0.074 | 0.018 | 0.054 | 0 |
| ***kernel_adi*** | 0.155 | 0.166 | 0.109 | 0.128 | 0.048 | 0.012 | 0.026 | 0.001 |
| **Mean** | 0.169 | 0.175 | 0.147 | 0.120 | 0.059 | 0.016 | 0.042 | 0.000 |

not have evolution operators such as crossover and mutation. Several major weaknesses of PSO include the proneness to getting trapped in local optimum especially in high-dimension design spaces, and the relatively low convergence rate during the iterative process. SA is a probabilistic technique aiming to approximate the global optima, which ignores past experiences and searches solutions to some extent hinging on randomness, thus not always reliable.

With respect to the comparison between AC and PG, PG generally provides better solutions, especially in the large cases such as the four real-case applications evaluated in this work. The reason is that the larger size the application has, the more difficult it is for the critic network in AC to accurately approximate state-value functions, which induces negative effects on the policy learning process. Notably, since the solutions generated by PG do not always dominate those by AC, we include both of them into the toolbox of IronMan, aiming to make RLMD a more powerful DSE engine.

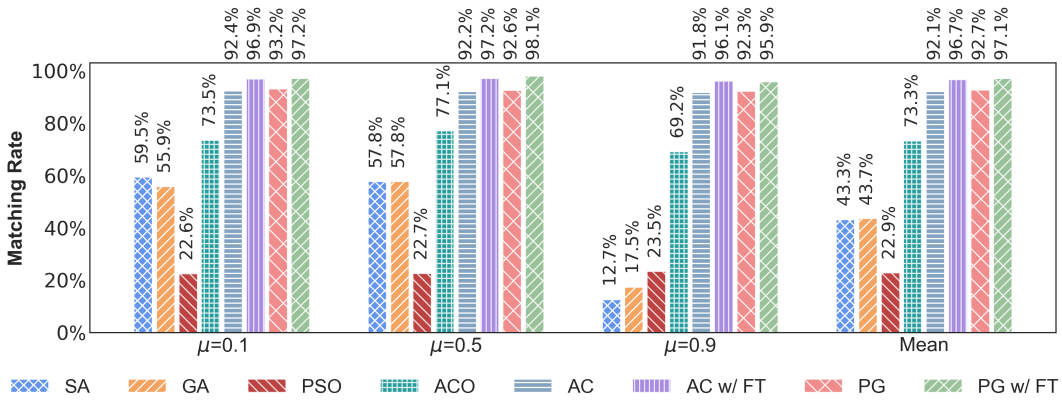


Figure 5.13: Matching rate of discrete DSP constraints, compared among SA, GA, PSO, ACO, IronMan (which applies AC or PG), and IronMan with FT. Here, three settings of $\mu$ are considered, and under each setting there are 323 discrete DSP constraints on the same four real-case applications, *gemm*, *kernel_2mm*, *spmv*, and *kernel_adi*. The average matching rate of each technique is the arithmetic mean on total 969 constraints.

**Solutions under user-specified constraints.** To further showcase that IronMan is capable to perfectly satisfy constraints without sacrificing latency, we specify different DSP constraints in a discrete manner. Among three different settings of $\mu$ and four real-case applications per setting, which totally makes up 969 DSP constraints, IronMan meets at least 92.1% of the cases and can further improve to 97.1% by FT; whereas SA, GA, PSO, and ACO only meet the constraints for 43.3%, 43.7%, 22.9% and 73.3% of the cases, respectively. Specifically, for the solutions that exceed DSP constraints, SA, GA, PSO, and ACO consume 1.42×, 1.55×, 1.48×, and 1.35× of target DSPs, respectively; on the other hand, RLMD and RLMD with FT consume significantly fewer DSPs, even when the constraints are not satisfied, which are 1.29× and 1.18× of target DSPs when applying AC, and 1.31× and 1.13× of target DSPs when applying PG.

### Execution Time

Table 5.3 shows the execution time of SA, GA, PSO, ACO, and IronMan (with and without FT) on real-case applications. It is noteworthy that neither HLS nor im-

Table 5.3: Execution time of SA, GA, PSO, ACO, and RLMD on for real-case benchmarks. Note that the running time of AC and PG during inference is similar, and the arithmetic mean is reported. The RLMD-FT further includes 500 episodes of training for FT.

| **Execution time (s)** | **SA** | **GA** | **PSO** | **ACO** | **RLMD** | **RLMD w/ FT** |
|---|---|---|---|---|---|---|
| ***gemm*** | 3758 | 5402 | 4074 | 5518 | 37 | 6048 |
| ***kernel_2mm*** | 3393 | 4117 | 3593 | 4552 | 27 | 5202 |
| ***spmv*** | 4292 | 6386 | 4264 | 5477 | 24 | 4641 |
| ***kernel_adi*** | 4229 | 4494 | 4017 | 5335 | 20 | 4424 |

plementation runtime is included in the reported execution time, since with the help of GPP, all the mentioned approaches do not need to invoke the entire design flow during the optimization process. HLS and implementation process are only invoked once after these approaches generate their solutions. In this sense, it is fair to only compare the execution time of each approach itself.

During inference, i.e., being applied on real applications, IronMan only takes a few seconds for prediction and solution generation. Vitis HLS takes tens of minutes to synthesize the C/C++ code, and takes up to hours to get the exact resource usage after implementation. The SA, GA, PSO, and ACO also take hours in average, because they struggle to exactly or closely meet the DSP constraints, and cannot transfer prior knowledge across different DFGs or DSP constraints. The FT for RL agent can balance between a quick solution using the pre-trained model (denoted as RLMD in Table 5.3) and a longer yet better one for a particular DFG (denoted as RLMD w/ FT in Table 5.3), which is optional and the number of episodes is adjustable based on users' requirements.

## 5.5 Conclusion

IronMan is an end-to-end framework, aiming to help HLS tools generate higher-quality solutions under user-specified constraints, or to perform more flexible DSE to

provide Pareto solutions that are not currently supported by HLS tools. IronMan is equipped with a GNN-based performance predictor GPP, an RL-based DSE engine RLMD, and a code transformer. Independently, GPP achieves high prediction accuracy, reducing prediction errors by 5.7× in timing predictions and 10.9× in resource usage predictions, compared with HLS tools; RLMD outperforms GA, SA, PSO, and ACO by 16.0% $\sim$ 29.5% in terms of LUT utilization, and by 7.6% $\sim$ 16.5% in terms of CP timing. Integrally, IronMan finds solutions that are within user-specified constraints over 96% of the cases, more than twice the number of cases handled by meta-heuristic-based techniques and with a significant speedup. These results demonstrate the great potential of applying GNN and RL for HLS optimization.

# Chapter 6

# Deterministic Policy Gradient for Workload Placement Optimization

In this chapter, we exploit deep deterministic policy gradient (DDPG) to optimize workload placement on multi-chip many-core systems, which is scalable to large systems with thousands of cores and can handle different connection topologies without requiring topology-specific knowledge.

With the rapid evolution of ML workloads, specialized architectures and accelerators have emerged, ranging from those optimized for CNNs (e.g., ShiDianNao [213], Eyeriss [214], and SCNN [215]) to those designed for general-purpose DNN acceleration (e.g., DaDianNao [216], Cambricon-x [217], EIE [218], TPU [219], and DNPU [220]). Existing DNN systems often diversify in performance, accuracy, and power requirements, which is prohibitively costly to build a dedicated accelerator/architecture for each target. Therefore, multi-chip many-core (MCMC) neural network systems, which assemble a number of cores into one chip and further interconnect these chips, are attracting increasing attention. These MCMC systems, from conventional technology such as SpiNNaker [221], TrueNorth [222], Loihi [223], Tianjic [224, 225], Simba [226], to emerging technology such

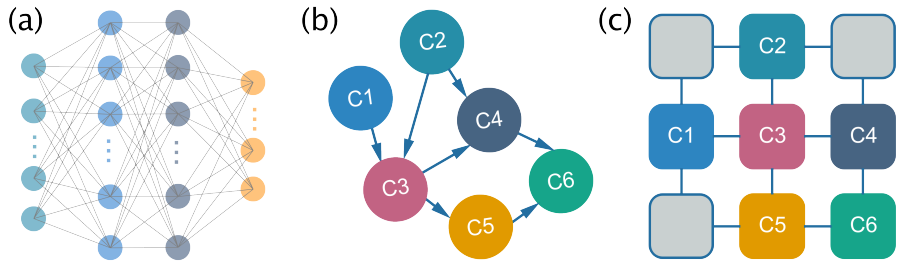


Figure 6.1: NN mapping: (a) the original NN; (b) the NN partitioned into logic cores; (c) logic cores placed onto physical cores.

as PUMA [227] with memristors, provide high parallelism benefited from decentralized execution, and can be scaled to very large systems with reasonable fabrication costs.

Usually, there are two major steps to map an application or a neural network (NN) model to a many-core system. In the first step, the computational graph is partitioned into small groups that are compatible to the computation capability of each core [224, 226, 228], in which we refer these small groups as **logic cores**, since some of them are logically connected with demand of communication and they are not yet placed on physical chips (see Figure 6.1(a) and Figure 6.1(b)). Then in the second step, these logic cores are placed onto **physical cores** – such process is defined as the **core placement** (see Figure 6.1(c)). As MCMC systems scale up, communication costs would be a concern, and workload partitioning and placement heavily impact the efficiency of on-chip and off-chip communication [222, 226, 229, 230]. Targeting the first step, several studies improve workload partitioning to reduce the required communication between logic cores. For example, Urgeses et al. [229] present a partitioning methodology to optimize network traffic for spiking neural networks on neuromorphic many-core platforms; HyPar [231] searches a partition that minimizes the total communication of DNNs on an accelerator array. Targeting the second step, heuristic-based methods are employed to map applications onto 2D-mesh network-on-chip (NoC) architectures [232, 233, 234, 235, 236, 237, 238].

However, there are two principal issues still unresolved in the core placement step. First, these previous approaches all target general-purpose many-core systems within *a single chip*, whereas in decentralized MCMC systems, the communication-related problem is caused by not only the demand for communication among different cores but also *the non-uniform and hierarchical on/off-chip communication capability*. Second, the *scalability* is a concern with these heuristic-based methods, since they are typically designed to handle systems with tens of cores and have limited DSE capabilities. Finding an optimal core placement is an *NP-hard* [239] problem and the search space of this problem *grows factorially* with the system size.

To this end, we propose an RL-based method to optimize workload placement on MCMC systems. This method performs a series of trials (i.e. placements) to understand which logic core should be placed on which physical core so that the overall latency can be optimized. The specific algorithm leveraged is DDPG [240], since the deterministic policy gradient can be estimated much more efficiently than the usual stochastic policy gradient, leading to a faster training process. We summarize our contributions as follows:

- **Problem formulation.** We consider DNN inference in MCMC systems and implement a hierarchical pipeline (i.e., a block-by-block streaming pipeline for intra-frame dataflow and a stage-based pipeline for inter-frame dataflow). Given the weight-stationary dataflow on the spatial MCMC architecture, we formulate the core placement optimization problem.

- **Core placement optimization.** We propose an RL-based method that utilizes DDPG to automatically optimize core placement, which is capable to handle systems with thousands of cores. The employment of deterministic policy gradient enables more efficient training, particularly in large action spaces. The proposed RL agent adopts CNNs to extract spatial features of different placements.

- **Evaluation.** We evaluate our proposed method on multiple workloads: AlexNet [241], VGG16 [242], and ResNet50 [243]. On the geometric average, it achieves 50.5%, 38.4%, 18.6% reduction in the overall latency and improves the throughput by 1.99×, 1.61×, 1.22×, compared with sequential placement, random search, and simulated annealing, respectively.

- **Release of domain expertise.** Our proposed method can automatically optimize core placements by leveraging the communication properties of different system configurations, without requiring any domain-specific knowledge.

This chapter is organized as follows: Chapter 6.1 introduces the background of DNN workloads, MCMC architecture, and related work; Chapter 6.2 describes the problem formulation and the proposed DDPG-based core placement optimization approach; Chapter 6.3 presents experiment setup, baselines, and analysis of experimental results; Chapter 6.4 concludes this chapter.

## 6.1 Background and Related Work

We provide a brief introduction to the background of DNN workloads, the MCMC architecture, and related work.

### 6.1.1 DNN Workload

There are multiple variants of DNNs, including MLP, CNN, RNN, and so on. As illustrated in Figure 6.2, a convolutional (CONV) layer can be considered as a seven-dimensional nested loop on input activations (IA), weights (W), and output activations (OA), with the batch size $B$, the height $H$ and the width $T$ of OA, the number of output channels $K$, the number of input channels $C$, the height $R$ and the width $S$ of the weight

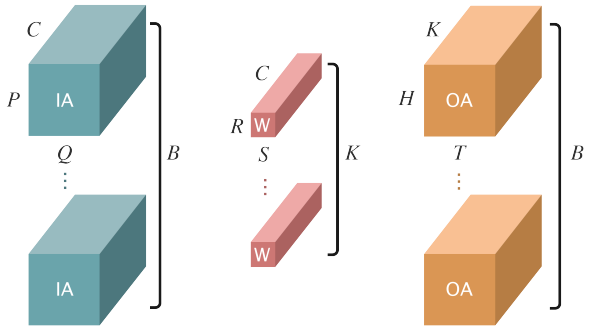


Figure 6.2: The seven-dimensional nested loop of convolutional layers, on input activations (IA), weights (W), and output activations (OA).

kernel. Similar formulations can also be applied to FC layers, which are widely used in MLPs and are essential components in DNNs.

### 6.1.2 Multi-Chip Many-Core Architecture

MCMC architectures, which are broadly employed to build up neuromorphic systems, arise with the era of cognitive computing that demands systems capable of processing massive amounts of multi-sensory data. Among the issues to be solved with top priority in these systems, real-time operation, low-power consumption, and scalability are those attracting the most attention, and thus parallel architectures working in a decentralized way are developed. There are several notable examples. SpiNNaker [221], which can model up to one billion neurons and one trillion synapses, integrates 18 ARM cores per chip and is able to scale to a system with 65536 chips. The TrueNorth chip [222] from IBM organizes 4096 neurosynaptic cores by 2D mesh, containing one million digital neurons and 256 million synapses; multiple TrueNorth chips can be further interconnected to build complex TrueNorth systems. Loihi [223] from Intel also utilizes the 2D mesh topology to comprise 128 neuromorphic cores and three embedded x86 processor cores on a single chip, and off-chip communication interfaces are used to connect other chips.

As the variety of DNN workloads increases and the performance, energy, and power targets diversify in different workloads, the concerns previously discussed in neuromor-

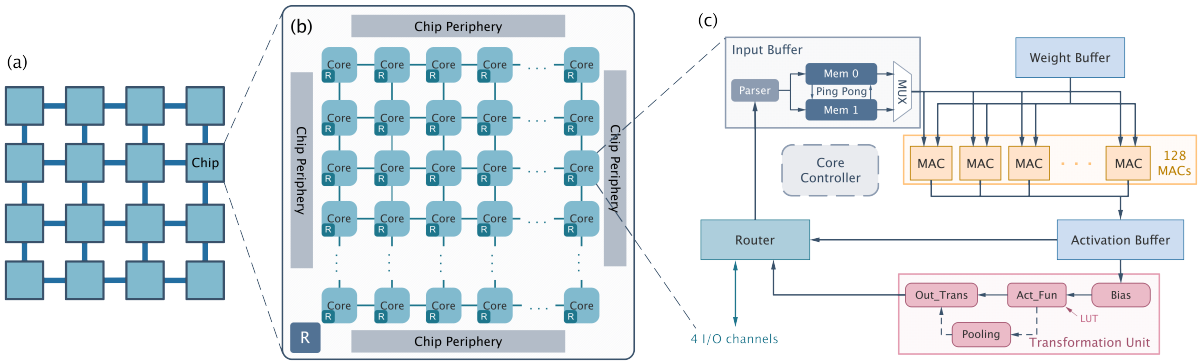


Figure 6.3: Illustration of a typical multi-chip many-core neural network architecture: (a) the multi-chip system, (b) the many-core chip, and (c) one single core.

phic systems also appear in the design of deep learning accelerators, and it is prohibitively costly to design a dedicated accelerator for each target of each workload. One potential resolution is to employ the multi-chip-module-based (MCM-based) integration. For example, Simba [226] is a scalable deep learning inference accelerator with an MCM-based architecture. In Simba, it is noticed that the disparity in latency and bandwidth between on-chip and on-package (off-chip) communication leads to significant latency variability across chiplets. To mitigate the inter-chiplet communication overheads, Simba optimizes workload partitioning and data placement by using a random search algorithm to select good mappings and placements.

In terms of the integration of both neuromorphic primitives (e.g. spiking neural networks) and DNNs, there is the Tianjic chip [224, 225], a MCMC architecture providing a hybrid platform towards artificial general intelligence. The Tianjic chip, consisting of 156 functional cores, shows significant improvement in both throughput ($1.6\times$ to $10^2\times$) and power efficiency ($12\times$ to $10^4\times$) compared with the GPU.

In Figure 6.3, we take Tianjic as an example to illustrate the typical MCMC architecture. Usually, multiple chips (e.g. $4\times4$ in Figure 6.3(a)) can be interconnected through off-chip links such as low-voltage differential signaling (LVDS) [244], SerDes [245], and ground-referenced signaling (GRS) [246, 247]. As shown in Figure 6.3(b), each chip in-

cludes an array of functional cores arranged by a 2D mesh NoC, an on-chip router for off-chip communication and essential chip peripherals. Figure 6.3(c) details the micro-architecture of each core, which leverages parallel multiplier-and-accumulator (MAC) units for efficient and flexible computation and contains peripheral processing circuits, such as an input buffer, a weight buffer, an activation buffer, a transformation unit, a core controller and a router. The input buffer provides input activations for MACs, where the ping-pong buffer scheme is used to decouple writes by the router and reads by MACs. The MACs conduct most of the computation, multiplying the input activations read from the input buffer with the weights stored in the distributed weight buffer to implement vector-matrix multiplications (VMMs). The activation buffer is used to buffer either intermediate activations or results that do not need to go through the transformation unit. The transformation unit is responsible for adding bias, non-linear activation functions, possible pooling operations, and generating output activations, and it finally sends the results to the router. The core controller manages the overall timing sequence and whether to enable these MACs or the transformation unit.

The MCMC architecture is essentially a spatial architecture, since there is no off-chip DRAM and all weights must be stored on chip, which is different from common deep learning accelerators. As such, it often uses a weight-stationary dataflow: weights remain in the weight buffer of each core and are reused across iterations, while new input activations are injected at each time phase.

### 6.1.3 Related Work

We review previous studies from two major categories: (1) mapping computation onto many-core systems, and (2) applying deep RL to optimize system latency.

### Mapping Computation onto Many-Core Architecture

A series of investigations in mapping applications onto 2D mesh NoC architectures has been conducted by applying various heuristic-based techniques. They mainly target minimizing communication energy consumption [232, 233, 234], reducing the total traffic loads and the average network hop count [235], or optimizing network latency [236, 237, 238]. There are four major differences between our work and the previous studies. First, these previous approaches all focus on general-purpose many-core architectures in a single chip. In contrast, we give attention to decentralized MCMC systems for NN workloads, where the communication related issue is caused by not only the demand for communication among different cores but also the non-uniform and hierarchical on/off-chip communication capability. Second, scalability is another challenge with these heuristic-based methods, since they mainly handle systems with tens of cores and the computation complexity grows drastically [232] as systems scale up. In contrast, our RL-based approach is capable to deal with systems with thousands of cores. Furthermore, previous work relies on topology-specific knowledge of 2D mesh NoCs, such as geometric features and communication characteristics, while our proposed method can work in a topology-agnostic manner.

### Device Placement Optimization with RL

In recent years, there is a surge in demand on computational resources in terms of training and inference of NNs with bigger models and larger batch sizes. One prevalent solution is to employ a heterogeneous distributed system with a mixture of different hardwares, with one instance of using the combination of CPUs and GPUs. In this scenario, the device placement refers to the process of mapping the computational graph of NNs onto hardware devices. Although computation partitioning and placement decisions are

usually made by human experts, there are still several concerns: first, expertise in both NNs and hardware architectures is required; second, these decisions are often based on simple heuristics and intuitions, which do not scale well or cannot produce optimal results, especially for complicated networks. To this end, Mirhoseini et al. [248] propose an RL-based method for device placement optimization, which uses a sequence-to-sequence RNN model as the parameterized policy to generate placements. This work manually groups operations and then places these groups onto devices, and later they develop a hierarchical end-to-end model by making the manual grouping process automatic [249]. In both of their work, network parameters are trained by policy gradients via the REINFORCE [50] algorithm. Spotlight [250] employs the proximal policy optimization (PPO) [251] to achieve better training speed and uses the softmax distributions to represent the policy. They further propose Post [252], which integrates PPO with cross-entropy minimization to acquire theoretically guaranteed optimal efficiency. Placeto [253] uses graph embeddings to encode the structure of computational graphs and exhibits good generalizability to unseen NNs, but having high computation costs.

## 6.2 Approach

We present a detailed problem formulation and describe our proposed DDPG-based approach for core placement optimization in MCMC systems.

### 6.2.1 Formulation of Core Placement Optimization

For simplicity and clarity, we consider the spatial mapping with a weight-stationary dataflow for DNN inference.

### Mapping Neural Networks to Logic Cores

Taking advantages of model parallelism, there have been several different DNN tiling techniques [215, 216, 219, 254] proposed to partition weights in the spatial mapping, based on which we partition DNN weights uniformly along the input channel $C$ and the output channel $K$. Figure 6.4 illustrates the uniform partitioning of CONV and FC layers. In the decentralized many-core system, outputs of VMM cores will be delivered to other cores for cross-core partial-sum reduction, referred to as vector-vector-accumulation (VVA), if handling with large NNs. We decouple the execution of VMM and VVA to different cores in order to ease the timing implementation.

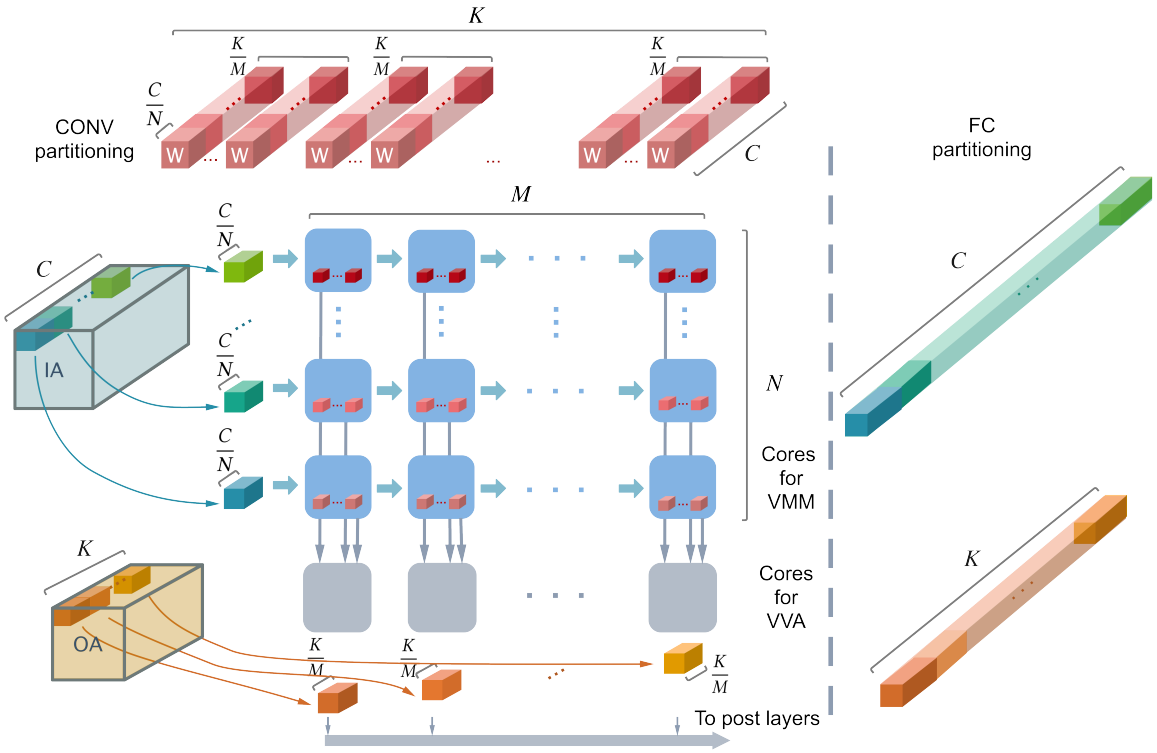


Figure 6.4: The uniform partitioning of CONV or FC layer, where the red tensors on the top represent weights (W), the green tensors in the middle represent input activations (IA), and the orange tensors at the bottom represent the output activations (OA).

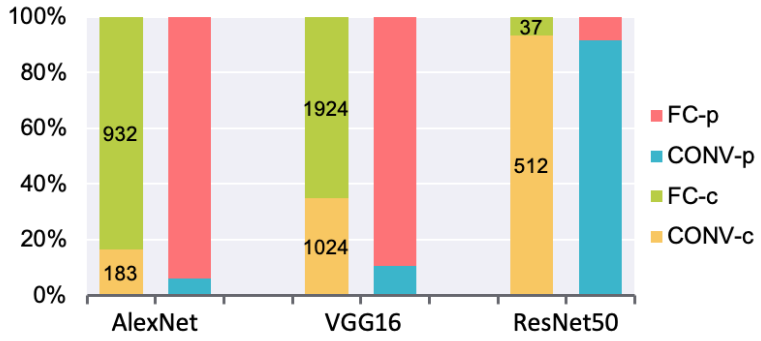


Figure 6.5: The breakdown of parameters (denoted with -p) and logic cores (denoted with -c) for CONV and FC layers in different models, with the number of logic cores marked for each model.

We further optimize the uniform partitioning by two steps: first, we balance the computation required on each core, to avoid over-busy or idle cores; second, we consider the trade-off between the exploitation of computation parallelism and the communication/synchronization costs. Figure 6.5 shows the breakdown of logic cores for different models. Since CONV layers are often bound by computation while FC layers are often bound by memory, more logic cores are assigned for CONV layers to balance the computation.

### Core Placement

Consider a set of logically connected cores consisting of $Z$ logic cores $\{C_1, C_2, ..., C_Z\}$, and a set of $D$ available physical cores $\{V_1, V_2, ..., V_D\}$ connected in a specific topology (where $Z \leq D$). A placement $\mathcal{P}$=$\{p_1, p_2, ..., p_Z\}$ is an assignment of a logic core $C_i$ to a physical core $p_i$, where $p_i \in \{V_1, V_2, ..., V_D\}$ and $\forall i \neq j$, there is $p_i \neq p_j$.

In each single frame, it is possible to implement a streaming pipeline across multiple CONV layers to take advantage of inter-layer parallelism because each convolution operation only needs part of input activations. In contrast, for FC layers one output activation cannot be generated until all input activations are ready, indicating that there only ex-

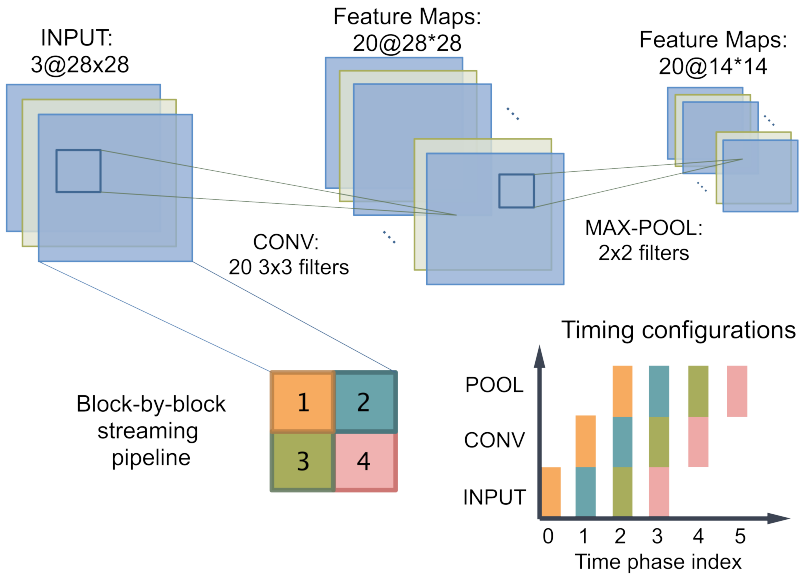


Figure 6.6: Example of the block-by-block streaming pipeline execution and its corresponding timing configurations.

ists intra-layer parallelism. Based on this execution difference, we consider a hierarchical pipeline execution.

For the inter-frame execution, a stage-based pipeline is used to decouple the computation of CONV and FC layers, leveraging better parallelism. Accordingly, we place the logic cores for CONV and FC layers in different regions of the MCMC system, and *optimize their core placement processes separately* by masking unused regions.

Then in the intra-frame execution, instead of the row-by-row streaming pipeline [255] in which logic cores have more and more idle time as layer propagates, we employ the block-by-block streaming pipeline for CONV layers to make better utilization of resources. As depicted in Figure 6.6, we showcase a block-by-block streaming pipeline and its corresponding timing configurations by an example of the MNIST dataset [256], where each input frame is divided into four blocks. In this example, there is a three-stage streaming pipeline, with one time phase for one pipeline stage; in order to guarantee system functionality, the time phase, which contains both the computation latency as well as

the communication latency, should be long enough to cover the pipeline stage that consumes the maximum latency. Similar analogy is used in FC layers, where the pipelined execution is applied layer by layer.

Assume there is an $\mathcal{F}$-stage streaming pipeline for intra-frame execution, we use $T(k|\mathcal{P})$ to denote the latency of the $k$-th pipeline stage for a given placement $\mathcal{P}$. By minimizing the maximum latency among all stages, the overall latency and throughput can be optimized. Therefore, the optimization goal is:

$$\mathcal{P}^* = \underset{\mathcal{P}}{\operatorname{argmin}} \{L(\mathcal{P})\} \tag{6.1}$$

where $L(\mathcal{P}) = \max_k \{T(k|\mathcal{P})\}$ for $k = 1, 2, ..., \mathcal{F}$.

### 6.2.2 Core Placement with Deep Deterministic Policy Gradient

Figure 6.7 presents the overview of the RL-based core placement optimization. The agent attempts to learn an optimal core placement to minimize the overall latency, and the environment gives feedback to the agent by different rewards to encourage or punish the agent according to its behaviors. Through interactions with the environment, the agent is able to learn and figure out the optimal policy.

We build the core placement problem as a Markov decision process. At the beginning of each trial, no assignment has been generated and all physical cores are available. At each time step $t$, with the observation of currently available physical cores and unplaced logic cores, which is referred to as the *state* $s_t$ in the state space $\mathcal{S}$, placement of a couple of logic cores will be generated, which is referred to as the *action* $a_t$ in the action space $\mathcal{A}$. With this action $a_t$, corresponding physical cores are occupied by these assigned logic cores, and the state $s_t$ is updated to the state $s_{t+1}$. The placement of logic cores is generated sequentially according to the index and the reward is provided at each time step.

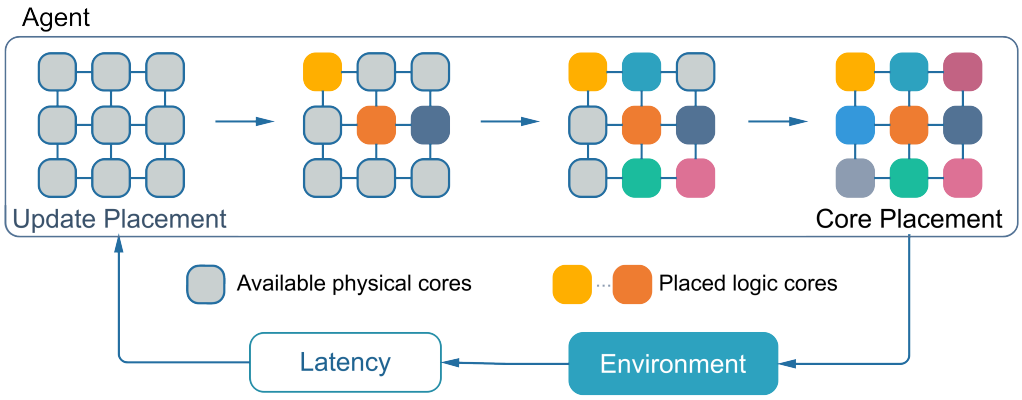


Figure 6.7: Overview of the RL-based core placement optimization.

It is notable that from our simulation results, different placement orders have little influence on learning performance, as the agent can adjust its behaviors during interactions with the environment, mitigating the effects caused by different orders. When all logic cores have been placed onto physical cores, the overall performance of this placement is measured by the maximum latency among all pipeline stages, i.e., $L(\mathcal{P})$, which is then used to derive the final reward of this placement. These rewards, together with information from states, actions, and state-action values, are combined to train the agent and update following placements.

### Representations of Core Placement Optimization

The mathematical representations of state, action, and reward of core placement optimization are detailed as follows.

- *Representation of Core Placements (i.e., the states).* Among most MCMC architectures, 2D mesh topology is the mainstream for both intra-chip and inter-chip interconnect. We mainly consider the communication characteristics when optimizing core placement, and thus we prefer the matrix representation of the placement, which is simpler and more intuitive. The state $s_t$ is represented by a 2D matrix to encode the current placement status, including the information required by the

agent to make decisions, as shown in the upper part of Figure 6.8. In this illustration, a $3 \times 3$ chip array with $2 \times 2$ cores per chip is represented by a $6 \times 6$ matrix, where the available physical cores are denoted by zero and occupied physical cores are denoted by the indexes of their assigned logic cores. In general, the state of the current placement on an MCMC system composed of a $\text{row}_{chip} \times \text{col}_{chip}$ chip array with $\text{row}_{core} \times \text{col}_{core}$ cores per chip can be denoted as a $(\text{row}_{chip} \times \text{row}_{core})$-by-$(\text{col}_{chip} \times \text{col}_{core})$ matrix.

- *Representation of Assigning Placements (i.e., the actions).* Given that the current core placement is uncompleted, the action is defined as assigning a placement of $z$ unplaced logic cores, which is encoded as $[x_1, y_1, x_2, y_2, ..., x_z, y_z]$, with $(x_i, y_i)$ representing the physical coordinate on which a logic core will be placed.

- *Representation of the Reward Function.* We empirically find that defining the reward at the time step with a completed placement as $r_t = \sqrt{\mathcal{B}} - \sqrt{L(\mathcal{P})}$, where $\mathcal{B}$ is the latency of the best placement found by the random search, makes the learning process more robust. With this definition, placements that result in better latency are encouraged by positive rewards, while placements that result in worse latency are penalized by negative rewards. For those time steps at which one placement is not completed, the reward is defined as $r_t = 0$.

**Deterministic Policy Gradient.**

Policy gradients have been broadly applied under different RL scenarios, where the basic idea is to directly parameterize the policy via a probability distribution $\pi_\theta(s, a) = \mathbb{P}(a|s; \theta)$ that stochastically takes the action $a$ given the state $s$ according to the param-

eters $\theta$. If we define the discounted state distribution [207] by

$$\rho_\pi(s') := \int_{\mathcal{S}} \sum_{t=1}^{\infty} \gamma^{t-1} \mathbb{P}(s_t = s' | s_0 = s, \pi) \mathbb{P}(s_0 = s) \mathrm{d}s, \tag{6.2}$$

then the expected return can be expressed as

$$\begin{aligned} \mathcal{J}(\pi_\theta) &= \mathbb{E}_{s\sim\rho_\pi, a\sim\pi_\theta} \left[ \sum_{k=0}^{\infty} \gamma^k r_{k+1} \right] \\ &= \int_{\mathcal{S}} \rho_\pi(s) \int_{\mathcal{A}} \pi_\theta(s,a) Q_\pi(s,a) \mathrm{d}a\mathrm{d}s, \end{aligned} \tag{6.3}$$

where $Q_\pi(s,a)$ is defined in Equation (2.3) and the discount factor $\gamma \in (0,1]$.

In order to maximize the expected return of a stochastic policy, the corresponding stochastic policy gradient algorithm should update the parameters $\theta$ by performing gradient ascent on the expected return, i.e., adjusting the parameters $\theta$ in the direction of $\nabla_\theta \mathcal{J}(\pi_\theta)$, where

$$\begin{aligned} \nabla_\theta \mathcal{J}(\pi_\theta) &= \nabla_\theta \int_{\mathcal{S}} \rho_\pi(s) \int_{\mathcal{A}} \pi_\theta(s,a) Q_\pi(s,a) \mathrm{d}a\mathrm{d}s \\ &= \int_{\mathcal{S}} \rho_\pi(s) \int_{\mathcal{A}} \pi_\theta(s,a) \frac{\nabla_\theta \pi_\theta(s,a)}{\pi_\theta(s,a)} Q_\pi(s,a) \mathrm{d}a\mathrm{d}s \\ &= \mathbb{E}_{s\sim\rho_\pi, a\sim\pi_\theta} \left[ \nabla_\theta \log \pi_\theta(s,a) Q_\pi(s,a) \right]. \end{aligned} \tag{6.4}$$

In our work, instead of the stochastic policy, we give attention to the deterministic policy [240]. We propose to train a deterministic policy $\mu_\theta(s) : \mathcal{S} \rightarrow \mathcal{A}$, which is a deterministic mapping from the current placement status $s_t$ to the action $a_t$ – the placement assignment of unplaced logic cores. With the deterministic policy, the core placement

process can be optimized by maximizing

$$\begin{aligned} \mathcal{J}(\mu_\theta) &= \mathbb{E}_{s\sim\rho_\mu}\left[\sum_{k=0}^{\infty}\gamma^k r_{k+1}\right] \\ &= \int_{\mathcal{S}} \rho_\mu(s) Q_\mu(s,\mu_\theta)\mathrm{d}s. \end{aligned} \tag{6.5}$$

Then the deterministic policy gradient is derived as

$$\begin{aligned} \nabla_\theta \mathcal{J}(\mu_\theta) &= \nabla_\theta \int_{\mathcal{S}} \rho_\mu(s) Q_\mu(s,\mu_\theta)\mathrm{d}s \\ &= \int_{\mathcal{S}} \rho_\mu(s) \nabla_\theta \mu_\theta(s) \nabla_a Q_\mu(s,a)|_{a=\mu_\theta(s)} \\ &= \mathbb{E}_{s\sim\rho_\mu}\left[\nabla_\theta \mu_\theta(s) \nabla_a Q_\mu(s,a)|_{a=\mu_\theta(s)}\right]. \end{aligned} \tag{6.6}$$

From the practical perspective, the cardinal reason for applying deterministic policy gradient rather than stochastic policy gradient is that the stochastic policy gradient should be estimated by the integration over both the state space and the action space, as shown in Equation (6.4); while the deterministic policy gradient only needs to integrate the state space as in Equation (6.6), indicating that it can be estimated more efficiently and leads to a faster learning process, especially for a large action space, which is our case.

From a practical perspective, the main reason for applying deterministic policy gradient rather than stochastic policy gradient is that the stochastic policy gradient needs to be estimated by the integration over both the state space and the action space, as shown in Equation (6.4). By contrast, the deterministic policy gradient only needs to integrate over the state space, as in Equation (6.6), which can be estimated more efficiently and leads to a faster learning process, especially for a large action space, which is the case in our study.

We employ the off-policy deterministic actor-critic (OPDAC) [240], which consists of

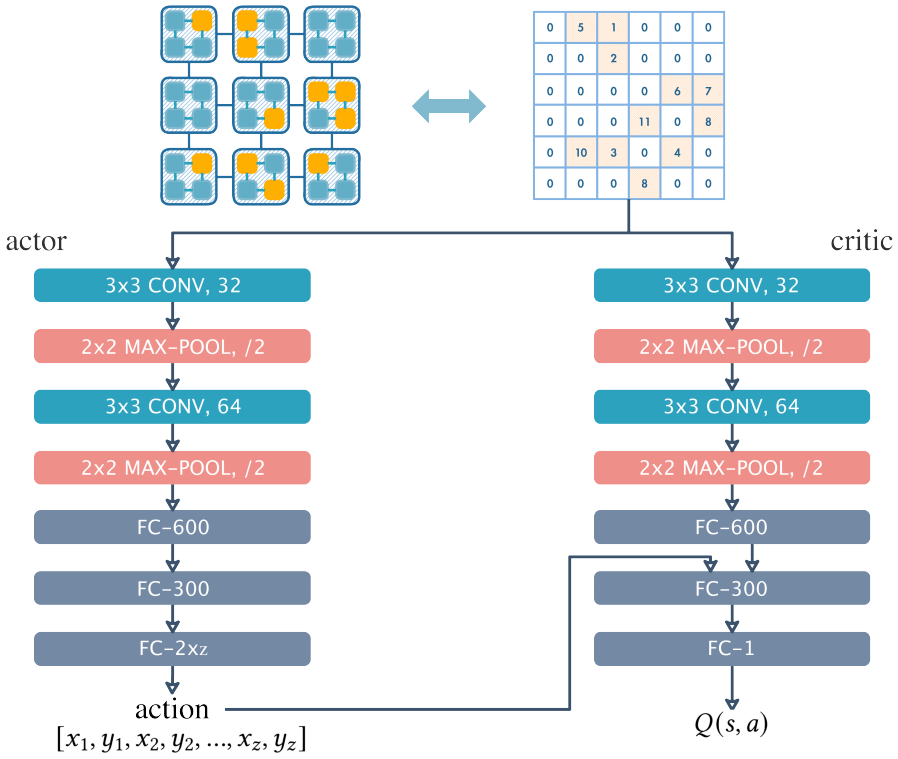


Figure 6.8: DNN structure of the RL-based agent: the actor, and the critic.

two components: the *critic* and the *actor*. The critic estimates the action-value function $Q_w(s, a) \approx Q_\mu(s, a)$ by adjusting parameters $w$ based on Q-learning, and the actor learns the deterministic policy $\mu_\theta(s)$ by ascending the gradient of the action-value function.

To improve the sample efficiency of the learning process, we apply the experience replay taking advantages of past experiences, which is implemented by a replay buffer storing tuples $(s_t, a_t, r_{t+1}, s_{t+1})$ from history trajectories. To sufficiently explore the large search space, we add Ornstein-Uhlenbeck noise [257] to the action space, which is multiplied by a fading factor as the training process proceeds.

### DNN Structure of the RL-based Agent

The structure of the RL-based agent is depicted in Figure 6.8, where both the actor and the critic have similar network structures. The input to these DNNs is the current

state of the placement being predicted, which includes physical cores either currently available or already assigned with logic cores. Then the actor outputs the action in vector, and the critic generates the state-action value in scalar. Since the state-action value is a function of both the current state and the action being taken, the output of the actor is merged to the critic after its first FC layer. We employ CONV layers followed with max-pooling layers to extract spatial features of various placements, because there are some similarities between the core placement analysis and image analysis, on which CONV layers usually perform well. The local response normalization is applied after each pooling layer, and the batch normalization is applied after each FC layer. The activation function is ReLU for all layers, except for the output of the actor, which uses *tanh* to bound actions to the size of MCMC systems. Since the outputs of the actor network are in continuous values, we apply the *floor* function to derive placement locations, i.e., finding the closest integers that are no larger than the outputs. If there is a contradiction between the currently being placed core and an already placed core, the current core will be placed on the position that has the minimum Manhattan distance to its originally intentional position. If there are multiple available candidates, we choose the first one found.

The network parameters are learned by Adam optimizer based on the estimation of Equation (6.6), which is computed by sampling a minibatch of size $\mathcal{K}_{mb}$ from the replay buffer, leading to the updates of parameters as follows:

$$\delta_i = r_{i+1} + \gamma Q_w(s_{i+1}, \mu_\theta(s_{i+1})) - Q_w(s_i, a_i), \tag{6.7}$$

$$w_{t+1} = w_t + \alpha_w \cdot \frac{1}{\mathcal{K}_{mb}} \sum_{i=t_1}^{t_{\mathcal{K}_{mb}}} \delta_i \nabla_w Q_w(s_i, a_i), \tag{6.8}$$

**Algorithm 2:** DDPG for core placement optimization.

Initialize parameters $\theta$ for the actor and $w$ for the critic;
Initialize the episode counter $i \leftarrow 0$;
Initialize the best core placement $\mathcal{P}_{best} \leftarrow \mathcal{P}_{baseline}$;
**while** $i < episode_{max}$ **do**
  $t \leftarrow 0$ ; `// The time step counter.`
  Initialize state $s_t \leftarrow$ an empty placement;
  **while** $t < step_{max}$ **do**
    Perform action $a_t$ based on policy $\mu_\theta(s_t)$;
    Get updated placement $s_{t+1}$;
    **if** *all logic cores have been placed* **then**
      Receive the reward $r_t = \sqrt{\mathcal{B}} - \sqrt{L(s_{t+1})}$;
      Add $(s_t, a_t, r_{t+1}, s_{t+1})$ into replay buffer;
      **if** $L(s_{t+1}) < L(\mathcal{P}_{best})$ **then**
        $\mathcal{P}_{best} \leftarrow s_{t+1}$;
      **end**
      Clear state $s_{t+1} \leftarrow$ an empty placement;
    **else**
      Receive the reward $r_t = 0$;
      Add $(s_t, a_t, r_{t+1}, s_{t+1})$ into replay buffer;
    **end**
    Update $\theta$ and $w$ according to Equations (6.7)-(6.9);
    $t \leftarrow t + 1$;
  **end**
  $i \leftarrow i + 1$;
**end**
**return** $\mathcal{P}_{best}$;

$$\theta_{t+1} = \theta_t + \alpha_\theta \cdot \frac{1}{\mathcal{K}_{mb}} \sum_{i=t_1}^{t_{\mathcal{K}_{mb}}} \nabla_\theta \mu_\theta(s_i) \nabla_a Q_w(s_i, a_i)|_{a=\mu_\theta(s)}, \tag{6.9}$$

where $\alpha_w$ and $\alpha_\theta$ are learning rates of the critic and the actor, respectively, and $i \in \{t_1, ..., t_{\mathcal{K}_{mb}}\}$.

The entire procedure of core placement optimization with deep deterministic policy gradient is summarized in Algorithm 2.

## 6.3 Experiment

We present the experiment setup, baselines, and the analysis of the results.

### 6.3.1 Experiment Setup

We build an in-house simulator for the typical MCMC architecture illustrated in Figure 6.3. The overall system consists of a $4 \times 4$ chip array with $16 \times 16$ cores per chip, with the off-chip interconnect assumed as GRS [246, 247]. Generally, the routing is based on the minimal path, with X-Y routing for both NoC and off-chip communication. Although all chips are functional in the MCMC system, different workloads may occupy different number of chips/cores, since in these spatially weight-stationary mappings, the number of cores consumed is kind of proportional to the model size. Configuration parameters are summarized in Table 6.1, which are collected from existing MCMC architectures [224, 225, 226, 246, 247]. As for hyperparameters in our RL-based approach, the learning rates of the actor and the critic are set as $\alpha_\theta = 0.0002$ and $\alpha_w = 0.001$, with the discount factor $\gamma = 0.98$. In each epoch, the actor predicts 30 placements and the size of minibatch is $\mathcal{K} = 64$.

We consider DNN workloads of AlexNet [241], VGG16 [242] and ResNet50 [243], and evaluate the latency when the batch size is one and the throughput when the batch size is much larger. The overall latency is derived according to the latency of each time phase that is measured by summing up the computation latency (i.e. the cycles required for computation) and the communication latency. As described in Chapter 6.2.1, we place the logic cores for CONV and FC layers in different regions of the MCMC system, and *optimize their core placement processes separately.*

Table 6.1: Simulation configuration parameters.

| | | |
|---|---|---|
| System | Number of Chips | $4 \times 4$ |
| | Off-chip Interconnect | GRS |
| | Off-chip Interconnect Bandwidth | 100GB/s/chip |
| Chip | Number of Cores | $16 \times 16$ |
| | Technology | UMC 28-nm HLP |
| | NoC Interconnect Bandwidth | 64GB/s/core |
| | Core Frequency | 400MHz |
| Core | Weight Buffer Size | 64KB |
| | Input+Activation Buffer Size | 64KB |
| | Number of MACs | 128 |
| | MAC Width | 8b |
| | Input/Weight Precision | 8b |
| | Partial-sum Precision | 32b |

### 6.3.2 Baselines

Our RL-based approach (denoted by DDPG) is evaluated with the following placement methods.

- *Sequential placement* (denoted by BS): logic cores are placed sequentially along with the indexes of physical cores (first chip index, then core index).

- *Random search* (denoted by RS): one million placements are sampled randomly, and the best placement found during the random search is selected.

- *Simulated annealing* [15] (denoted by SA): SA is commonly applied for design exploration, with the procedure detailed in Algorithm 3. The cooldown factor is set as 0.99; the initial temperature $T_0$ and the ending temperature $T_{end}$ are chosen according to the application such that around one million placements would be searched; and the neighborhood function $\mathcal{N}(\mathcal{P}_{current})$ indicates that the placement of 1% of logic cores in $\mathcal{P}_{current}$ will be randomly changed.

**Algorithm 3:** Simulated annealing for core placement optimization

```
1  Randomly generate initial core placement P_current ← P_0;
2  Initialize the best core placement P_best ← P_0;
3  Initialize temperature T ← T_0;
4  while T > T_end do
5      iter ← 0;
6      while iter < iteration_max do
7          Select a placement P_new ∈ N(P_current);
           /* A neighbor placement to current placement.          */
8          if L(P_new) < L(P_current) then
9              P_current ← P_new;
10             if L(P_new) < L(P_best) then
11                 P_best ← P_new;
12             end
13         else
14             Δ = L(P_new) − L(P_current);
15             Accept P_current ← P_new with probability ℙ = e^{−Δ/T};
16         end
17         iter ← iter + 1;
18     end
19     T ← α × T;
       /* 0 < α < 1, the cooldown factor.                         */
20 end
21 return P_best;
```

### 6.3.3 Analysis of Core Placements Optimized by DDPG

Figure 6.9 compares the latency and the throughput of different core placement methods for AlexNet, VGG16, and ResNet50 workloads. DDPG achieves significant improvements among all considered workloads, especially for VGG16 that has the largest model size, where DDPG reduces the overall latency by 67.4%, 51.7%, 23.2%, and improves the throughput by 3.06×, 2.07×, 1.30×, compared with BS, RS, and SA, respectively. Generally, there is usually a larger optimization space for large models, because they are often mapped onto more logic cores, resulting in a larger search space, just as aforementioned that the search space of core placement optimization *grows factorially* with the system size. In addition, more conspicuous improvements are shown in FC layers than those in


Figure 6.9: Latency and throughput of different placement methods, both of which are normalized to BS (i.e., sequential placement).

CONV layers, since the inter-layer connections are denser in FC layers; one exception comes from the FC layers in ResNet50, whose small layer size leads to a relatively small search and optimization space. Furthermore, the communication demand usually relates to the size of feature maps, the number of input and output channels, and whether there exist bypass connections in the networks, indicating that the more complex the network structure is, the higher the communication demand is often required, and thus the more essential it is to conduct the core placement optimization. DDPG demonstrates stronger improvements in VGG16 and ResNet50, since they have more complex network structures compared to AlexNet. Considering all the workloads, on the geometric aver-

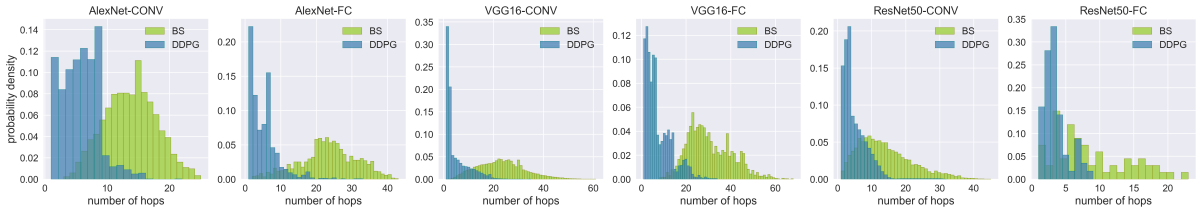


Figure 6.10: Hop distributions of BS and DDPG-based core placement optimization.

age, 50.5%, 38.4%, 18.6% reductions in the overall latency are achieved by DDPG, with the throughput improvement of 1.99×, 1.61×, 1.22×, compared with BS, RS, and SA, respectively.

Notably, in scenarios with extremely large search spaces, DDPG substantially outperforms SA. In SA, new placements at each time are randomly picked from the neighborhood of the current placement, and whether or not to accept a new placement is dependent solely on the latency or the objective function, ignoring past experiences and introducing unreliability to the search process. In contrast, DDPG proactively explores the search space. By learning from different rewards received during the exploration, DDPG extracts useful spatial features from various placements, to avoid defective placements and further encourage trials to approach the optimum. Through the leverage of experience replay, past experiences can be consolidated into the training process, thus stabilizing the overall learning and search process.

To show more insights of core placements found by DDPG, Figure 6.10 depicts the hop distributions before and after DDPG-based core placement optimization. The averaged hop distances in the CONV and FC layers are reduced by 2.5× and 4.5× for AlexNet, 4.6× and 4.3× for VGG16, and 2.9× and 2.8× for ResNet50, respectively. The geometric average reduction in hop distance is 3.5×. This indicates that long data paths are significantly shrunken and cores that are logically connected are tended to be placed in nearby regions, reducing the long travel time of data as well as removing potential

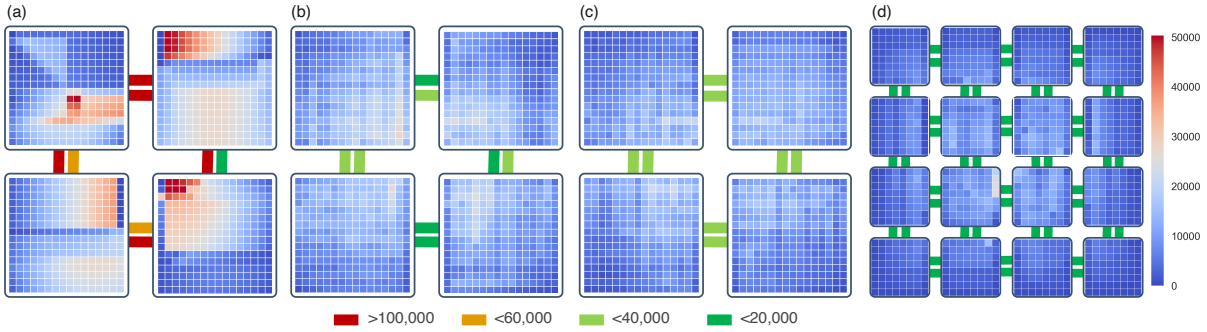


Figure 6.11: The distribution of the total number of packets transferred through each core per time phase, and the total number of packets delivered by each off-chip link per time phase, for core placements of VGG16-CONV: (a) placed by BS, (b) optimized by DDPG, (c) with doubled off-chip bandwidth optimized by DDPG, and (d) with fewer cores per chip optimized by DDPG.

congestions. Additionally, with the reduced hop distances, the active communication power consumption can also be implicitly reduced.

Figure 6.11(a) and (b) show the communication traffic of placements optimized by BS and DDPG. For BS, there are multiple extremely busy cores for on-chip communication and several off-chip links with heavy communication workloads; whereas after the optimization by DDPG, both on-chip and off-chip communication are balanced: unnecessary off-chip communication is minimized and moved to on-chip communication that usually consumes lower costs, and the traffic of busy cores is spread to relatively idle cores. DDPG ensures that the off-chip traffic is low enough to avoid congestion delay, thus improving the latency.

### 6.3.4 Strong Learning Capability of DDPG

Our proposed DDPG-based core placement optimization exhibits strong learning capabilities in efficiently utilizing various communication configurations and can be applied to other topologies such as 2D torus, HNoC [258] and dragonfly [259], in a topology-agnostic manner.

**Efficient Utilization of Communication Configurations**

To investigate the sensitivity of DDPG-based core placement optimization to the off-chip communication bandwidth, we adjust the off-chip bandwidth to 1.5× and 2.0× its original configuration, and apply core placement optimization under each configuration. It is undoubted that an increase in bandwidth should result in a reduction of latency, even if the original placement is not modified to account for these changed communication properties. In order to demonstrate the influence coming from the increased off-chip communication bandwidth, we fix placements that are optimized under the original configuration and only make changes in the off-chip communication bandwidth; as shown in Figure 6.12, there achieves less than 10% reduction in latency. Then in Figure 6.13, we compare the optimized placements found by DDPG and SA under each new configuration, to figure out their abilities to make use of different communication configurations. Obviously, more improvements are achieved by DDPG than those of SA: for CONV layers in VGG16, SA decreases the latency by 10% and 19%, while DDPG reduces the latency by 18% and 31%, with 1.5× and 2.0× off-chip bandwidth, respectively; for FC layers in AlexNet, SA decreases the latency by 4% and 8%, while DDPG reduces the latency by 12% and 17%, with 1.5× and 2.0× off-chip bandwidth, respectively. In both cases, DDPG demonstrates superior capabilities in identifying and leveraging the communication properties of the system, resulting in much beeter placements under the respective configurations Figure 6.11(c) shows the traffic of the placement optimized by DDPG with doubled off-chip bandwidth, where DDPG leverages the improved off-chip communication capability by subtly increasing the off-chip communication workloads and slightly alleviating the on-chip communication, compared to Figure 6.11(b).

We also make attempts to another case, where the number of cores per chip is decreased from $16 \times 16$ to $8 \times 8$ and so the number of chips is quadrupled. In this case,

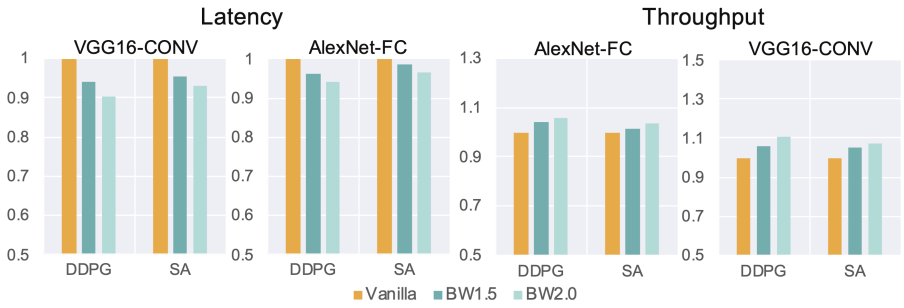


Figure 6.12: Latency and throughput of the placements optimized under the original (vanilla) configuration, with changing off-chip communication bandwidth.

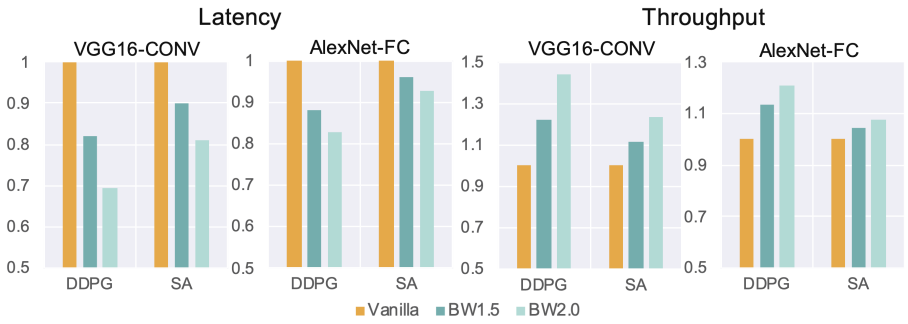


Figure 6.13: Latency and throughput optimized by DDPG and SA under each different off-chip communication bandwidth.

resources are sacrificed for performance, i.e., adding more communication resources to release the average communication burden on each off-chip link. It is worth noting that directly reducing the number of cores per chip in the absence of modifying the previously optimized placements may cause unpredictable effects. As shown in Figure 6.14, some placements may see performance gains, while others may experience performance degradation, which is mainly due to the possible disruption of spatial locality in communication. After core placement optimization under the new configuration, SA attains 22% and 4% reduction in latency, while DDPG reaches 39% and 24% reduction in latency, for CONV layers in VGG16 and FC layers in AlexNet, respectively, which is illustrated in Figure 6.15. DDPG optimizes core placement via trials and interactions with the environment to better understand and further leverage communication characteristics brought from different hierarchical structures; it can also make better utilization of the spatial locality in communication patterns of different workloads, where logic cores obtaining

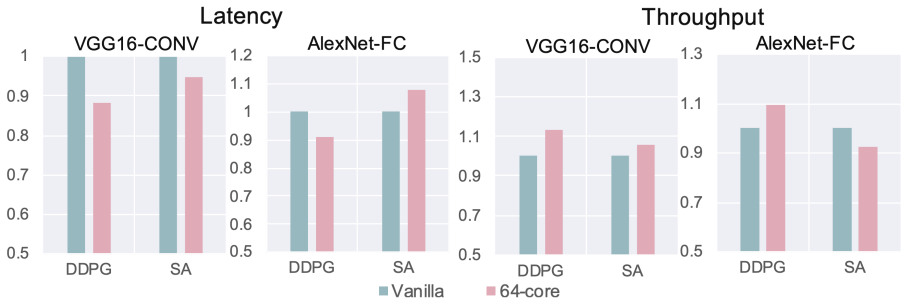


Figure 6.14: Latency and throughput of the placements optimized under the original (vanilla) configuration, with changing the number of cores per chip.

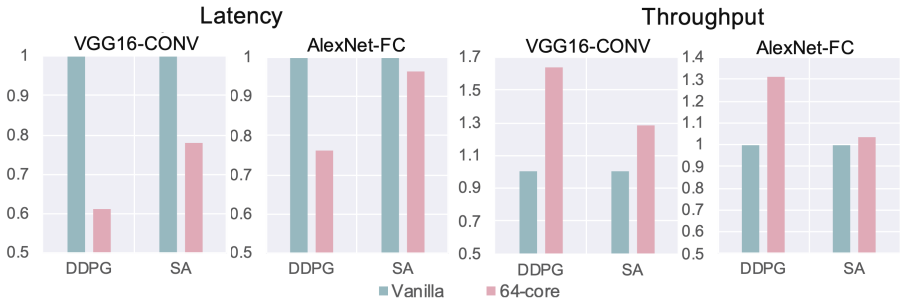


Figure 6.15: Latency and throughput optimized by DDPG and SA under each different number of cores per chip.

more connectivity are grouped more tightly. As displayed in Figure 6.11(d) that exhibits the communication traffic of the placement optimized by DDPG with $8 \times 8$ cores per chip, the off-chip communication is apparently reduced and balanced, with lightweight on-chip communication; central chips and cores are relatively busier, since packets from other cores may transit through them.

### Working in a Topology-Agnostic Manner

In addition to the 2D mesh, DDPG has great versatility to deal with other topologies, such as 2D torus, HNoC [258], and dragonfly [259]. We demonstrate this by building several small MCMC systems, since these topologies may have scalability issues: for 2D torus and HNoC, we use a $3 \times 3$ chip array with $2 \times 2$ cores per chip; for dragonfly, we use six chips with five cores per chip. All other configurations are set the same as those shown in Table 6.1, except for the weight buffer size and the input/activation buffer size,

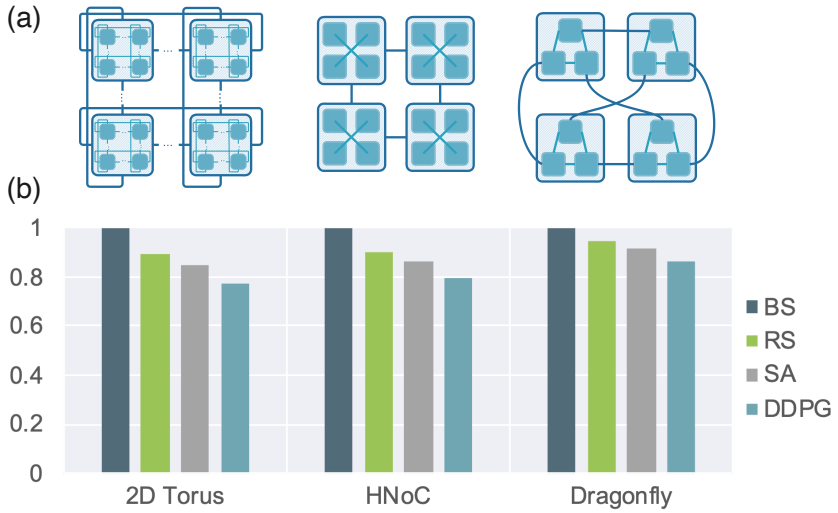


Figure 6.16: Latency of different placement methods for different topologies: (a) illustration of different topologies, and (b) latency normalized to BS.

both of which are selected as 16KB. A synthetic MLP with 600-467-124-103 structure is taken as the workload.

Figure 6.16 displays the latency of different core placement methods for different topologies. Even though SA already attains good performances, it is surpassed by DDPG: on the geometric average, DDPG achieves 19%, 12%, and 8% reduction in latency, compared with BS, RS, and SA, respectively. Through the leverage of CONV layers, DDPG is able to figure out spatial features aroused from different topologies, which is essential and beneficial for an optimized placement; through the leverage of past experiences, DDPG has a better understanding of both the system and the placement being predicted.

## 6.4 Conclusion

Workload partitioning and placement significantly impact communication efficiency, especially in decentralized systems. As MCMC systems scale, two major challenges arise in workload placement optimization: how to handle the non-uniform and hierarchical nature of on/off-chip communication capabilities, and how to ensure scalability to very

large MCMC systems Therefore, we propose an RL-based method that utilizes DDPG to automatically optimize workload placement, which is capable to handle MCMC systems with thousands of cores. We evaluate our proposed method on AlexNet, VGG16, and ResNet50, where on average DDPG reduces the overall latency by 50.5%, 38.4%, and 18.6%, and improves the throughput by 1.99×, 1.61×, 1.22×, compared with BS, RS, and SA, respectively. Our proposed method can automatically optimize core placements by leveraging the communication properties of different system configurations, without requiring any domain-specific knowledge.

# Chapter 7

# Graph Learning based Symbolic Reasoning for Large-Scale Boolean Networks

In this chapter, we leverage the message-passing mechanism in GNN computation to imitate conventional symbolic reasoning methods, enhancing their efficiency and scalability in large-scale Boolean networks (BNs) by making better use of modern computing power. Such reasoning process offers significant benefits in functional verification, logic minimization, datapath synthesis, malicious logic identification, and more.

Reasoning high-level abstractions (e.g., functional blocks) from bit-blasted BNs (e.g., unstructured gate-level netlists) has demonstrated its wide applications in improving functional verification efficiency [260, 261] and identifying malicious logics such as detecting hardware trojan and intellectual property infringement usage [262, 263]. In the era of globalization and democratization of integrated circuit (IC) development and fabrication, such reasoning is expected to bring broader impacts on hardware security, which is at the heart of modern computing systems: more than 40 percent of FPGA/ASIC projects

are working under safety-critical development process standards or guidelines [264].

Due to the optimization conducted by RTL synthesis tools, reasoning high-level abstractions such as functional blocks from unstructured or flattened netlists is extremely challenging, since hierarchy and module information is lost during multi-level logic minimization and technology mapping, which is also complicated by functional blocks overlapping and gate sharing. The problem goes further due to the explosion in runtime for large-scale BNs. Conventional reasoning approaches leverage structural analysis and functional propagation. Structural approaches either adopt shape hashing based on circuit topology to find structurally similar wires to form word-level abstractions [265], or rely on reference libraries to map sub-circuits with reference circuits [266]. Functional approaches focus on identifying functionally equivalent gates and wires by cut enumeration [267, 268]. The combination of structural and functional analysis [265, 267, 269] is more prevalent for efficient word-level abstraction and propagation. Despite the achieved success, the performance of these conventional approaches is restricted by **limited scalability** and **inefficient utilization of modern computing power**: (1) structural hashing is very time/memory-consuming for large BNs with billions of nodes; (2) functional propagations by symbolic evaluation are solver-ready but extremely expensive, in particular for *bit-blasted* non-linear arithmetic BNs; (3) all these algorithms do not effectively utilize modern computing power due to the difficulty of parallelism.

Motivated by the limitations of conventional approaches and the potentials of GNNs applied on circuit designs, we propose a graph learning-based symbolic reasoning framework to reverse engineer functional blocks from gate-level netlists, namely Gamora, which has **high reasoning accuracy**, **strong scalability** to BNs with billions of nodes, and **generalization capability** from simple to complex designs. Gamora employs a multi-task GNN to guarantee reasoning accuracy while simultaneously handling structural and functional information from BNs. Once well trained, Gamora becomes adept

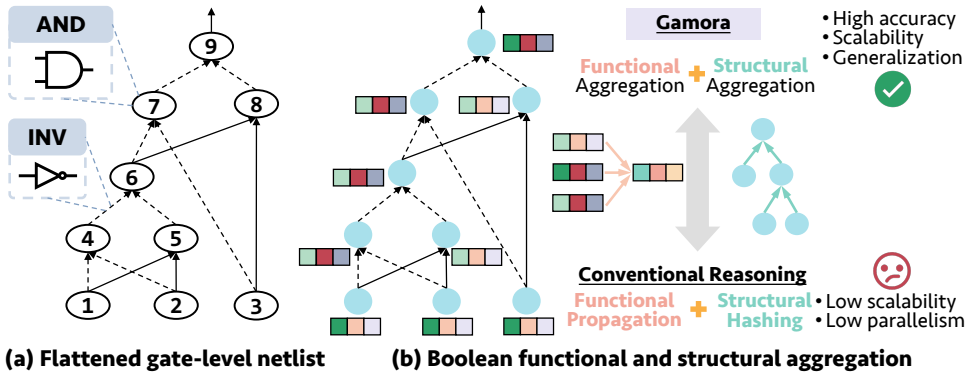


Figure 7.1: The inputs to GAMORA are flattened gate-level netlists, with each node as an AND gate and dashed edges as inverters. By encoding Boolean functional information as node features, GAMORA can simultaneously handle functional and structural aggregation, analogous to functional propagation and structural hashing in conventional reasoning but with strong scalability.

at generalizing to large-scale and complex BNs, leveraging the accelerated inference and parallel processing offered by modern computing systems. We summarize our contributions as follows.

- **Novel multi-task GNN for structure and function fusion.** The message passing mechanisms in GNNs enable simultaneous *Boolean functional* and *structural aggregation*, corresponding to the symbolic propagation and structural hashing in conventional reasoning methods, as shown in Figure 7.1. The multi-task setting allows knowledge sharing across different reasoning sub-tasks to guarantee high reasoning accuracy.

- **Billion-node scalability and parallelism.** We develop domain-specific techniques to compress node features, significantly reducing compute costs. The exploitation of graph learning draws better support from modern computing systems, such as GPU deployment, for scalability to large BNs and parallel execution.

- **Generalization capability.** Unlike many ML-based approaches that are trained with complex designs and infer on simpler ones, GAMORA can easily generalize from

simple to complex BNs and handle the reasoning complexity introduced by more advanced designs (such as Booth-encoded multipliers) and technology mapping.

- **Evaluation.** Regarding reasoning performance, GAMORA reaches almost 100% and over 97% reasoning accuracy for carry-save array (CSA) and Booth multipliers, respectively; after technology mapping, the reasoning accuracy is still over 92%. Regarding runtime and scalability, GAMORA can perform reasoning for large BNs with tens of millions of nodes/edges within one second, with a speedup of up to six orders of magnitude compared to the logic synthesis tool ABC [159].

- GAMORA is available at https://github.com/Yu-Utah/Gamora.

This chapter is organized as follows: Chapter 7.1 covers the basics of BNs, AIGs, and word-level abstraction, as well as our motivations; Chapter 7.2 presents how multi-task GNNs is applied in our proposed GAMORA to imitate conventional symbolic reasoning; Chapter 7.3 details experiment setup and evaluation on GAMORA in terms of reasoning accuracy, influence from design complexity, and runtime analysis; Chapter 7.4 concludes this chapter.

## 7.1 Preliminary and Motivation

### 7.1.1 Boolean Networks and And-Inverter Graphs

BNs are well-studied discrete mathematical models with broad applications in chemistry, biology, circuit design, formal verification, and more. A BN consists of a discrete set of binary variables, each of which is assigned a Boolean function taking inputs from a subset of these variables. The explosive growth in the scale of BNs has brought up increasing attention to analyzing their static and dynamic behaviors. For the purposes

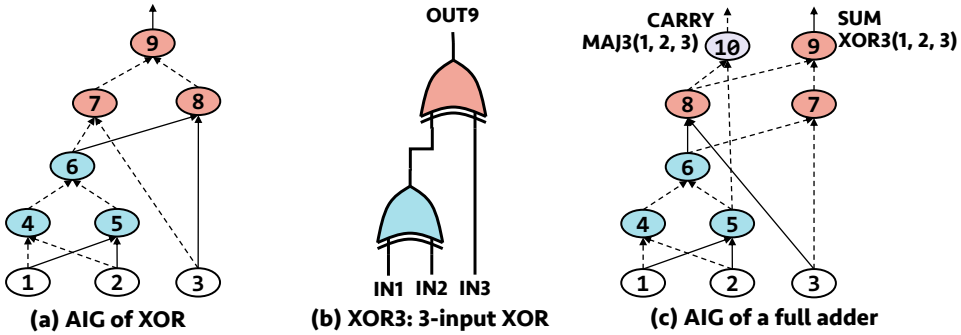


Figure 7.2: Netlists of XOR and a full adder. (a) AIG of XOR3 function. (b) XOR3 function: OUT9 = XOR3(IN1, IN2, IN3). (c) Full adder with a sum function (i.e., XOR3) and a carry-out function (i.e., MAJ3).

of synthesis and verification, a concise and uniform representation of BNs, consisting of inverters and two-input AND-gates, known as AIGs, has found successful use in diverse EDA tasks, since AIGs enable rewriting, simulation, technology mapping, placement, and verification to share the same data structure [270]. In an AIG, each node has at most two incoming edges; a node without incoming edges is a primary input (PI); primary outputs (POs) are denoted by special output nodes; each internal node represents a two-input AND function. Based on De Morgan's laws, any combinational BN can be converted into an AIG [159] in a fast and scalable manner.

In AIGs, cut enumeration can be used to detect Boolean functions. A feasible cut of node $n$ is a set of nodes in the transitive fan-in cone of $n$, whose truth value assignments completely determine the value of $n$. A cut is $K$-feasible if there are no more than $K$ inputs. Figure 7.2 depicts an example of reasoning XOR functions and full adders from AIGs. In Figure 7.2(a), the AIG has a 3-feasible cut of node 9 and a 2-feasible cut of node 6; after truth table computation, the functions of node 6 and node 9 are IN1$\oplus$IN2 and IN1$\oplus$IN2$\oplus$IN3, respectively. Thus, as shown in Figure 7.2(b), node 6 is an XOR2 function, and node 9 is an XOR3 function. Figure 7.2(c) shows a full adder bitslice, with the sum as an XOR function and the carry-out as a majority (MAJ) function. By pairing an XOR3 with a MAJ3 with identical inputs, a full adder bitslice can be extracted, which

is then aggregated for word-level abstraction.

### 7.1.2 Word-Level Abstraction

Word-level abstraction significantly reduces the complexity of large-scale BNs by grouping wires into meaningful words and keeping useful information related to control logic, which is widely applied in reasoning functional units from gate-level netlists [265, 267, 269]. Conventional word identification uses *structural shape hashing* and *functional bitslice aggregation*. Structural shape hashing assigns each edge in the BN a shape, which is defined as the directed graph constructed by the backward reachable nodes from this edge within certain depth/steps. Functional bitslice aggregation adopts functional matching to group functionally equivalent nodes and edges by cut enumeration. Typically, structural hashing and functional aggregation are iteratively propagated across neighborhood nodes using symbolic evaluation [265, 267, 269]. However, for large-scale BNs, structural hashing is memory-consuming; functional bitslice aggregation is not efficient due to the requirement of bit-blasting; the computation of symbolic evaluation is also expensive. Motivated by the **limited scalability** and the **difficulty of parallelism**, we propose to exploit **graph learning and GPU acceleration for highly scalable reasoning**.

### 7.1.3 GNNs for Circuit Reverse Engineering

Since BNs and circuit netlists are naturally represented as graphs, GNNs can be leveraged to classify sub-circuit functionality from gate-level netlists [271], predict the functionality of approximate circuits [272], analyze impacts of circuit rewriting on functional operator detection [273], and predict boundaries of arithmetic blocks [274]. Promising as they are, these approaches focus on graphs with tens of thousands of nodes, and con-

duct training on complex designs and inference on relatively simpler ones, in which the generalization capability from simple to complex designs is not well examined.

GNNs operate by propagating information along the edges of a given graph. The propagation along edges extracts structural information from graphs, corresponding to structural shape hashing in conventional reasoning; after encoding Boolean functionality into node features, neighborhood aggregation is analogous to functional aggregation in conventional reasoning. Thus, the inherent message-passing mechanism in GNNs enables simultaneously handling structural and functional information. Motivated by **the analogy between GNN computation and conventional reasoning**, we propose **a multi-task GNN for high-performance reasoning** to imitate exact reasoning algorithms, with **strong generalization capability** from simple to complex designs.

## 7.2 Proposed Approach

### 7.2.1 Overview

Figure 7.3(a) shows the overview of GAMORA. The inputs are flattened gate-level netlists in AIG format, without any micro-architectural or RTL information. These AIGs are generated by the logic synthesis tool ABC [159]. The goal is to exploit a multi-task GNN to reason high-level abstractions by performing node-level classification on AIGs, after which functional blocks (e.g., adders) can be extracted based on the annotated AIGs.

#### Case Study on Multipliers

Integer multipliers are indispensable to computationally intensive applications, such as signal processing and cryptography applications. Recent years also witness the strong

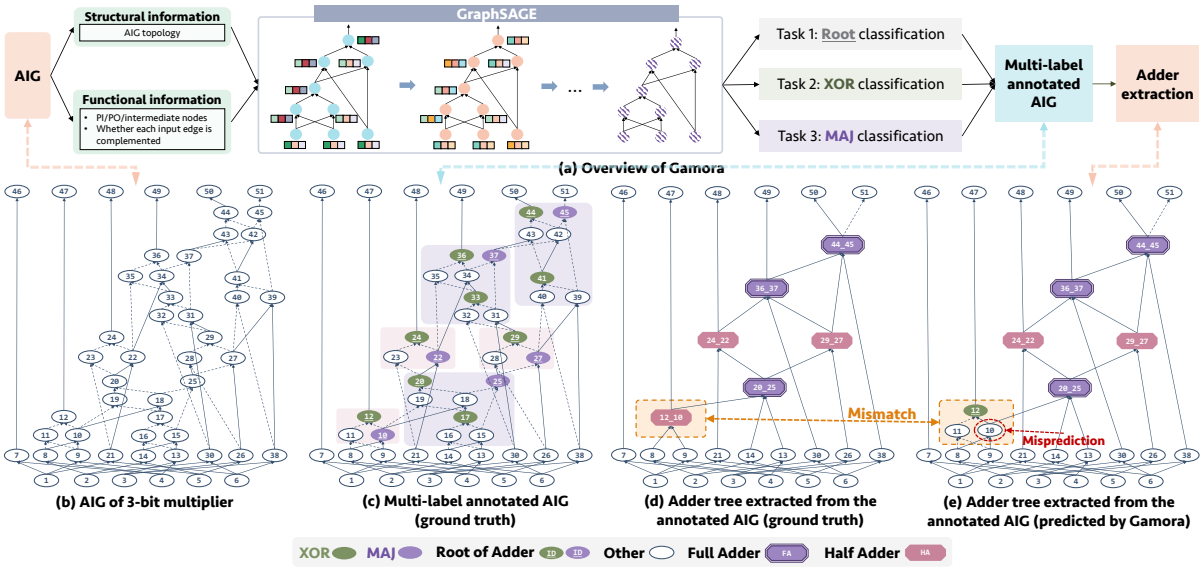


Figure 7.3: Overview of GAMORA. (a) GAMORA takes in flattened netlists in AIG format and performs multi-task node classification to reason the Boolean function of each node, after which the adder trees within multiplier netlists can be automatically extracted to improve the efficiency of word-level abstraction. (b) AIG of a 3-bit CSA multiplier after synthesis. (c) Annotated AIG with the Boolean function of each node, using the ground truth provided by ABC. (d) Adder tree extracted based on the *exact* reasoning, including three FAs and three HAs. (e) Adder tree extracted based on the reasoning performed by GAMORA.

demand for large integer multipliers in homomorphic encryption [275]. In general, formal verification for multipliers is challenging, especially for structurally complex designs such as Booth-encoded multipliers [260, 261, 276]. Symbolic computer algebra (SCA) has been successfully employed to verify a variety of integer multipliers [260, 261, 269, 277, 278], which relies heavily on detecting full adders (FAs) and half adders (HAs) in multiplier netlists. The state-of-the-art implementation in ABC framework [269] develops a fast algebraic rewriting approach to extracting adder trees from flattened multiplier netlists by detecting pairs of XOR and MAJ functions, which can handle large bitwidth multipliers (up to 2048-bit) but with extremely long runtime. Thus, targeting integer multipliers, we leverage GNNs to identify XOR and MAJ functions to extract adders from flattened netlists, which improves the efficiency of word-level abstraction from BNs and has strong

scalability enabled by GPU acceleration.

### 7.2.2 Multi-Task Learning for Boolean Reasoning

Boolean reasoning requires gathering structural and functional information from neighbor nodes, a process that can be imitated by the message-passing mechanism in GNNs. The task of reasoning high-level abstractions from flattened netlists, i.e., pinpointing adders from AIGs, involves a two-step procedure [265, 267, 269]: (1) detecting XOR/MAJ functions to construct adders, and then (2) identifying their boundaries. Therefore, we propose to apply multi-task learning (MTL) for Boolean reasoning to approach its nature, and the knowledge sharing across sub-tasks provides higher reasoning precision. Here details (1) how structural and functional information are fused in node embeddings, (2) how the two-step reasoning is formulated as a multi-task node classification, and (3) the post-processing after performing reasoning on each node in AIGs.

#### Fusing Structural and Functional Information

We leverage the message propagation and neighborhood aggregation in GNNs to generate the node embeddings of AIGs that simultaneously fuse structural and functional information. First, the structural information is distilled by passing node embeddings along edges that connect them. Second, the Boolean functional information can be encoded in node features. For each node, there are three node features represented in binary values denoting node types and Boolean functionality. The first node feature indicates whether this node is a PI/PO or intermediate node (i.e., AND gate). The second and the third node features indicate whether each input edge is inverted or not, such that AIGs can be represented as homogeneous graphs without additional edge features. These compressed node features not only encapsulate Boolean functionality of each node but

also enable high compute and memory efficiency. Figure 7.3(b) shows the AIG of a 3-bit CSA multiplier, in which the structural information is presented in the AIG topology and the functional information is encoded in node features. For example, node 1 is a PI with the feature vector [0, 0, 0]; node 7 is an internal node without negation on inputs, so the feature vector is [1, 0, 0]; node 17 has two inputs inverted, with the feature vector [1, 1, 1].

With the emphasis on generalization from simple to complex designs, the specific model employed is GraphSAGE [39]. Given a GraphSAGE model with $K$ layers, the node embeddings propagated between different layers are computed as follows:

$$\begin{aligned} h^k_{\mathcal{N}(v)} &\leftarrow \text{AGGREGATE}_k(\{h^{k-1}_u, \forall u \in \mathcal{N}(v)\}); \\ h^k_v &\leftarrow \sigma(\mathbf{W}^k \cdot \text{CONCAT}(h^{k-1}_v, h^k_{\mathcal{N}(v)})). \end{aligned} \tag{7.1}$$

Here, $\mathcal{N}(v)$ is the immediate neighborhood of node $v$; $\text{AGGREGATE}_k$ and $\mathbf{W}^k$ are the aggregation function and the weight matrix for layer $k$, respectively, where $\forall k \in \{1, ..., K\}$. After stacking $K$ layers, the structural and functional information within $K$-hop search depth is fused in the embedding of each node.

**Multi-Task Classification**

We identify the Boolean function of each node by a multi-task node classification to approach the nature of the problem: there are two steps involved in reasoning functional blocks from unstructured AIGs. The first step detects XOR and MAJ functions from AIGs, which will be used to construct adders. Since each XOR/MAJ function consists of multiple nodes in AIGs, only the root nodes of these functions are labeled as XOR/MAJ with other nodes marked as plain nodes. In addition to the exact XOR/MAJ functions, negation-permutation-negation equivalent functions are also labeled as XOR/MAJ. The

second step aims to automatically identify the boundaries of HAs and FAs, and thus we label roots (i.e. the sum and the carry-out functions) and leaves of each adder. Figure 7.3(c) shows a multi-label annotated AIG of a 3-bit multiplier, using the ground truth provided by ABC. Notably, one node can have multiple labels. For example, node 20 is labeled with XOR and the root of an adder; node 17 is labeled with XOR.

The MTL not only follows the intuition of this two-step reasoning but also exploits divide and conquer, since it is extremely hard for GNNs to reach high prediction accuracy with a single-task multi-label node classification. The employment of MTL enables knowledge sharing across sub-tasks and improves sample efficiency during training, which guarantees high reasoning performance. Specifically, the two-step reasoning is decoupled into three simpler classification tasks using generated node embeddings: *Task 1* classifies the roots and leaves of adders; *Task 2* and *Task 3* detect XOR and MAJ nodes, respectively. We use hard parameter sharing for MTL and the overall loss function $\mathcal{L}$ is shown as follows:

$$\mathcal{L} = \alpha \cdot \ell(\hat{y_1}, y_1) + \beta \cdot \ell(\hat{y_2}, y_2) + \gamma \cdot \ell(\hat{y_3}, y_3), \tag{7.2}$$

in which $\ell$ is the negative log-likelihood between predictions (i.e., $\hat{y_1}$, $\hat{y_2}$, and $\hat{y_3}$) and the ground truth (i.e., $y_1$, $y_2$, and $y_3$), and $\alpha$, $\beta$, and $\gamma$ are hyper-parameters to adjust the importance of each task. In our implementation, $\alpha = 0.8$ and $\beta = \gamma = 1$.

### Adder Tree Extraction from Multi-Labeled Graphs

After performing the multi-task node classification, we can recognize XOR, MAJ, and root nodes of adders. The XOR and MAJ pairs with identical inputs are matched to construct adders. The conversion from Figure 7.3(c) to 7.3(d) depicts the adder tree extraction. In Figure 7.3(c), the AIG has a set of XOR nodes $\mathbb{X} = \{12, 17, 20, 24, 29, 33, 36, 41, 44\}$ and a set of MAJ nodes $\mathbb{M} = \{10, 22, 25, 27, 37, 45\}$. After removing the

nodes that are not marked as adder roots, $\mathbb{X} = \{12, 20, 24, 29, 36, 44\}$. Given $\mathbb{X}$ and $\mathbb{M}$, node 12 is XOR3(8, 9, 0) and node 10 is MAJ3(8, 9, 0), a three-input XOR/MAJ function with node 8, node 9, and the constant zero as the inputs; node 20 is XOR3(10, 13, 14) and node 15 is MAJ3(10, 13, 14); this matching process continues until all six pairs of XOR and MAJ are generated, which are three FAs and three HAs, as shown in Figure 7.3(d).

Notably, GAMORA adopts graph learning to mimic the *exact* reasoning. In Figure 7.3(e), one HA cannot be automatically extracted due to the misprediction of node 10. Our evaluation indicates only several nodes near the least significant bit are always mispredicted because of their shallow neighborhood structure, which has a subtle impact on the efficiency of algebraic rewriting. By fusing structural and functional information into node embeddings and using MTL to approach the reasoning nature, GAMORA is expected to reach as close as possible to the *exact* reasoning precision.

## 7.3 Experiment

### 7.3.1 Experiment Setup

The AIG-based CSA and Booth multipliers are generated by the logic synthesis tool ABC [159], with the ground truth provided by the adder tree extraction command [269]. We consider two technology libraries: (1) the reduced standard-cell library *mcnc.genlib* (with gate input size <=3) from SIS distribution [279], and (2) ASAP 7nm technologies [170]. The GNN-based framework is implemented in Pytorch Geometric [150]. Two GraphSAGE models are developed for simple and complex design netlists: (1) a shallow 4-layer model with the hidden channel of 32 (for CSA multipliers w/ and w/o simple technology mapping), and (2) a deep 8-layer model with the hidden channel of 80 (for

Booth multipliers and after complex technology mapping). The generated node embeddings are passed to a shared linear layer with size of 32 and the ReLU activation function, followed by another linear transformation with softmax for each sub-task to perform node classification. Experiments are performed on a Linux host with AMD EPYC 7742 64-core CPUs and one NVIDIA A100 SXM 40GB GPU. *In general,* GAMORA *is trained on small bitwidth multipliers (typically less than 32-bit) and evaluated on large bitwidth multipliers (up to 2048-bit).*

### 7.3.2 Evaluation on Reasoning Performance

We evaluate the reasoning performance from three aspects: (1) how functional and structural information influences reasoning precision; (2) how design complexity affects model selection and training; (3) how technology mapping complicates the reasoning process and what domain insights can be derived to facilitate more accurate symbolic reasoning on complex BNs.

#### Reasoning Precision Analysis

Figure 7.4 illustrates how the reasoning performance on CSA multipliers is affected by different bitwidth multipliers for training, single/multi-task setting, and the employment of functional information. First, the larger bitwidth multiplier is adopted for training, the higher reasoning precision can be achieved, which typically converges after training with 8-bit multipliers. The main reason is for CSA multipliers, an 8-bit multiplier is able to provide a sufficient variety of structural properties, which can be learned and well generalized to larger multipliers by GAMORA. Second, the multi-task setting conspicuously outperforms the single-task counterpart, indicating that the knowledge sharing across multiple tasks greatly benefits the prediction accuracy of every single task. Third, there

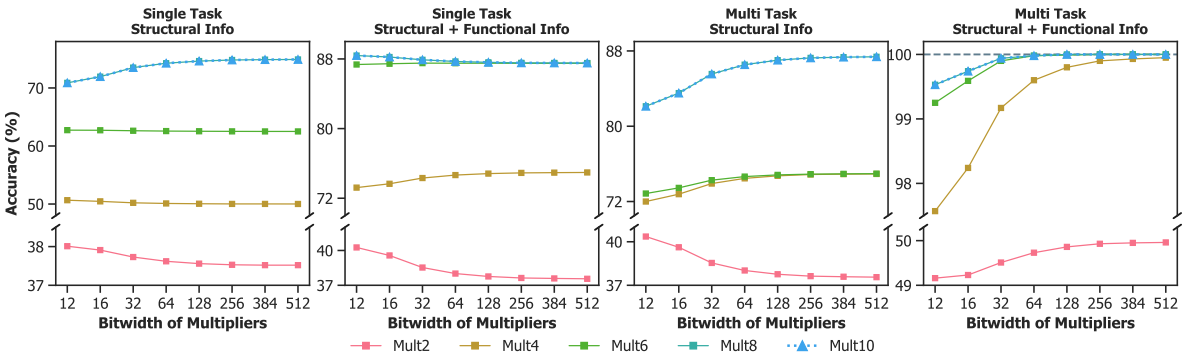


Figure 7.4: Sensitivity analysis on CSA multipliers with respect to (1) the bitwidth of multipliers for training (ranging from 2-bit to 10-bit), (2) single/multi-task, and (3) whether employing functional information.

is always a boost of accuracy when employing functional information for prediction, since identifying the role of each node relies on not only the surrounding structure but also the function of itself and its neighbors. The synergy of structural and functional information in GAMORA is analogous to the combination of structural hashing and functional propagation in conventional symbolic reasoning.

With the multi-task setting and simultaneously fusing structural and functional attributes, GAMORA achieves almost 100% prediction accuracy in symbolic reasoning for CSA multipliers. It is noted that several nodes near the least significant bit are always mispredicted due to their shallow neighborhood structure, as shown in Figure 7.3(e). This means the HA at the least significant bit cannot be automatically extracted, but can be easily corrected during post-processing.

### The Impact of Design Complexity

We analyze the impact from design complexity by evaluating the reasoning performance on radix-4 Booth-encoded multipliers, as shown in Figure 7.5. From the model selection aspect, as Booth multipliers generally have more complex structures, deeper models are necessary to characterize neighborhood structures and provide informative

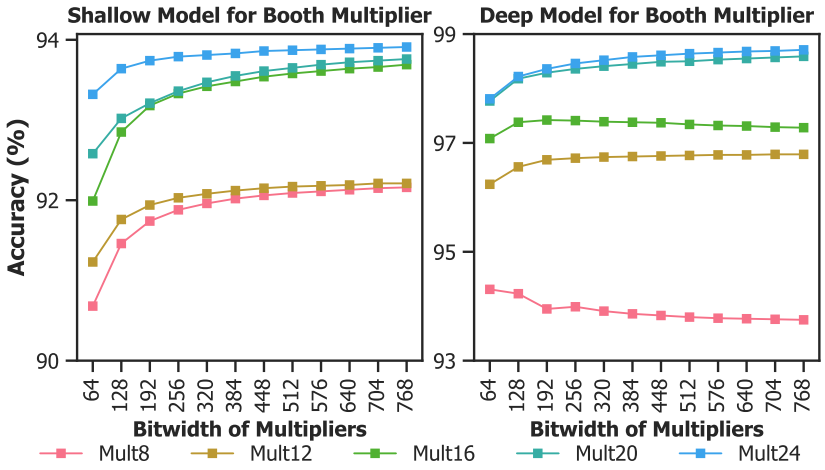


Figure 7.5: Evaluation on Booth multipliers with shallow and deep models.

node embeddings, ensuring high prediction accuracy. From the training aspect, larger multipliers (i.e., up to 24-bit Booth multiplier) are required for training such that adequate variety and representativeness of structural and functional characteristics are exposed to and well captured by GAMORA.

### The Impact of Technology Mapping

It is a known challenge that technology mapping can increase the complexity of formal reasoning on BNs [265, 267, 280]. Thus, we evaluate the performance of GAMORA with respect to different technology mapping options. The multipliers are mapped using the ABC standard-cell mapper (i.e., using the command `map`). Figure 7.6 depicts the reasoning performance on CSA and Booth multipliers after simple technology mapping [279] and ASAP 7nm technology mapping [170]. Specifically, the ASAP 7nm library contains 161 standard-cell gates, including multi-output cells such as the full adder cell, which significantly increases the complexity and irregularity of post-mapping netlists.

In the simple technology mapping case, the models trained before technology mapping demonstrate good generalization capability, still reaching over 99% and 92% predic-

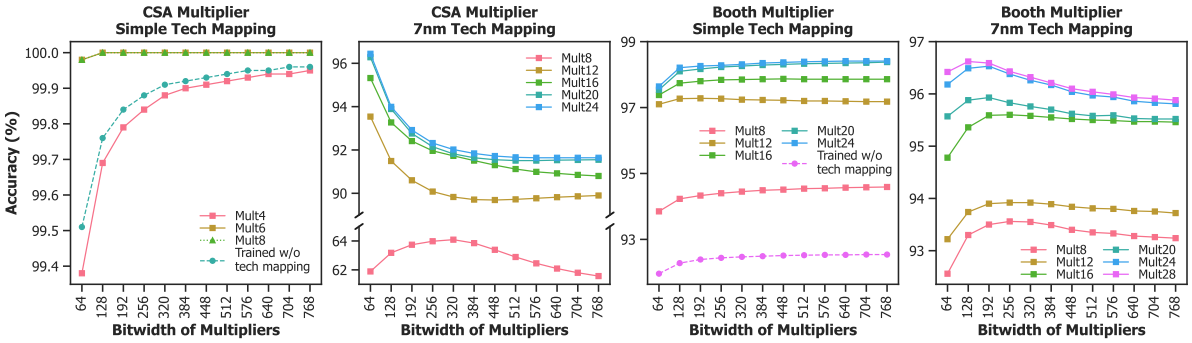


Figure 7.6: Evaluation on CSA and Booth multipliers, with simple and complex technology mapping.

tion accuracy for CSA and Booth multipliers, respectively; with retraining, comparable reasoning performance to those on original multipliers is achieved with similar sizes of training multipliers. The scenario is fairly different in the case of ASAP 7nm technology mapping, which employs a relatively complex technology library: first, the generalization capability is limited before and after technology mapping; second, the prediction accuracy slightly drops even with retraining; third, it is necessary to use large training multipliers to guarantee performance.

These observations imply several takeaways. First, the more complex technology library is applied, the more difficult it is for learning-based symbolic reasoning, since more complexity is involved both in AIG structures and the functionality of each node. This also implicates attributes related to the technology library should be included in node and edge features. Second, the capability to cope with intricate AIG netlists comes at the expense of more comprehensive training data. One underlying assumption of many supervised ML tasks is the training and testing data should be independent and identically distributed, which is governed by a fundamental principle called empirical risk minimization that provides theoretical performance bounds [281]. Thus, increasing the size of training data can envelop more knowledge of interested statistical properties, ensuring better generalization to testing data.

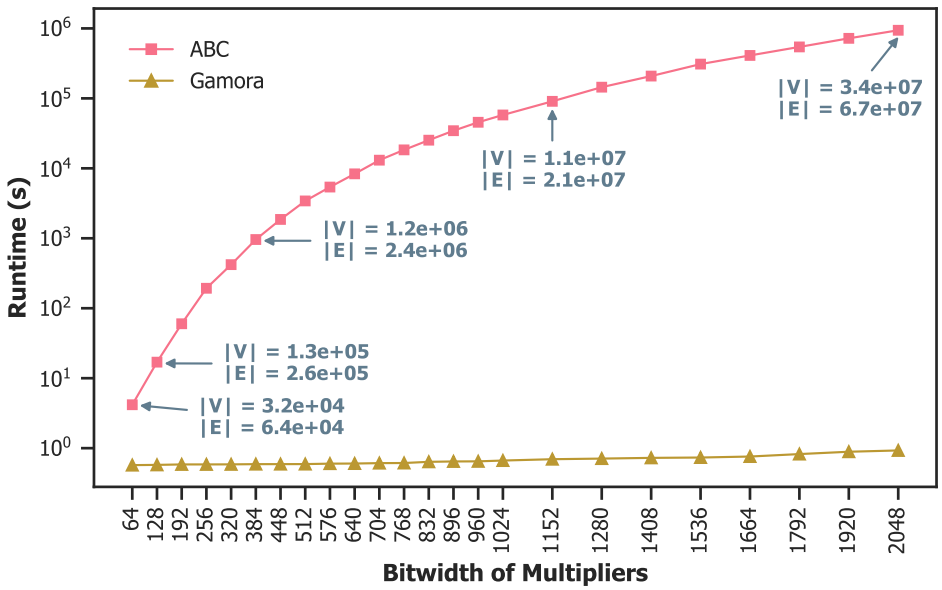


Figure 7.7: Runtime comparison between GAMORA and ABC. Note that the number of nodes $|V|$ and the number of edges $|E|$ are annotated for scalability analysis.

### 7.3.3 Runtime and Scalability Analysis

In addition to the high reasoning performance, we demonstrate the superiority of GAMORA by analyzing its runtime and scalability.

#### Runtime Complexity Analysis

Basically, the runtime only relates to the scale of AIGs, i.e., the number of nodes $|V|$ and the number of edges $|E|$. Figure 7.7 compares the runtime of GAMORA against ABC on CSA multipliers: for large designs such as a 2048-bit CSA multiplier with around 34 million nodes and 67 million edges, GAMORA attains a speedup of up to six orders of magnitude. This shows not only the great efficiency in symbolic reasoning enabled by graph learning but also the scalability to extremely large designs.

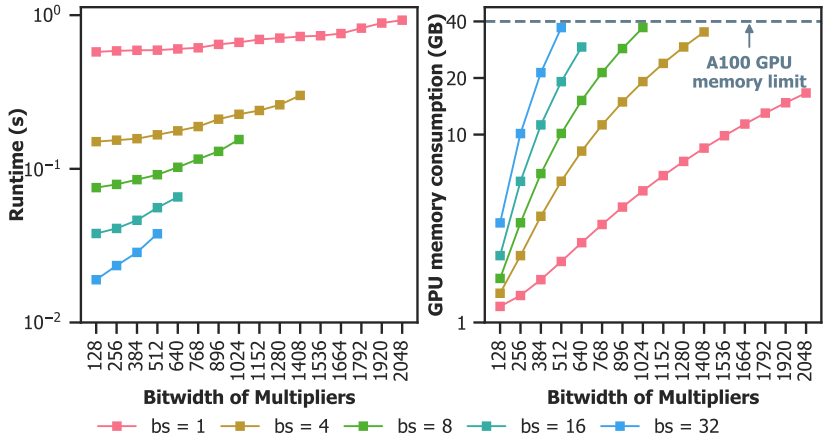


Figure 7.8: Average runtime and GPU memory consumption with batched reasoning, with the batch size denoted as bs. We currently focus on single-GPU implementation.

### Batched Reasoning with Single GPU

Figure 7.8 shows further acceleration allowed by batched reasoning. Currently, we focus on single GPU implementation, which limits the batch size by the GPU memory, and leave multi-GPU implementation as our future work to support larger batch processing. Even with a single GPU, there already reveal promising results and positive trends benefiting from parallel execution and GPU acceleration.

## 7.4 Conclusion

Reasoning high-level abstractions from bit-blasted BNs has benefited functional verification, logic minimization, datapath synthesis, malicious logic identification, etc. In this chapter, we propose a novel symbolic reasoning framework, GAMORA, which exploits GNNs to imitate structural hashing and functional aggregation in conventional reasoning approaches. Evaluation shows that (1) with the proposed multi-task GNN model, GAMORA offers **high reasoning performance** that reaches almost 100% and over 97% accuracy for CSA and Booth-encoded multipliers, which is still over 92% in finding func-

tional modules after complex technology mapping; (2) with GPU acceleration on graph learning, GAMORA has **strong scalability** to BNs with over 33 million nodes, with up to **six orders of magnitude speedups** compared to the state-of-the-art implementation in the ABC framework; (3) GAMORA also demonstrates **great generalization capability** from simple to complex designs, such as from small to large bitwidth multipliers, and from before to after technology mapping. GAMORA reveals the great potential of applying GNNs and GPU acceleration to speed up symbolic reasoning.

# Chapter 8

# Conclusion

## 8.1 Summary of Past and Current Contributions

The explosion of modern hardware complexity is challenging the optimality and scalability of conventional hardware development methodologies and EDA tools, resulting in long time-to-market as well as high capital and labor costs. Given the avidity toward productivity boost, it is highly expected to embrace more intelligence. Aiming to foster the virtuous cycle between ML and hardware, this dissertation focuses on **ML-empowered agile hardware development** and investigates how to integrate intelligence into scalable and effective hardware development flows through synergistic research across algorithms, architectures, and EDA. Specifically, the contributions can be briefly summarized as follows.

- **Fast and accurate design evaluation.** Targeting HLS and logic synthesis, we leverage the inherent graph structures of DFGs and circuits and explore how domain knowledge can be integrated into different GNN models, so that we can reconcile timeliness, accuracy, and generalization capability.

  - In Chapter 3, we answer why and how GNNs can be applied for HLS performance predictions [23]. We formally formulate HLS performance predictions using GNN, profile 14 state-of-the-art GNN models, and propose a knowledge-infused hierarchical GNN model incorporating domain knowledge to balance prediction accuracy and timeliness.
  - In Chapter 4, we innovatively employ multi-modal graph learning that simultaneously handles spatio-temporal information from hardware designs and logic synthesis flows to provide accurate and generalizable assessments of QoRs after logic synthesis [24].

- **Efficient and scalable design optimization.** We exploit the power of deep RL for faster, better, and more flexible optimization that can make the impossible possible and impel hardware design and EDA forward into the future.
  - In Chapter 5, we propose an end-to-end optimization framework, IronMan [25, 26], for fine-grained, flexible, and automated DSE in HLS. The primary goal is to provide either optimized solutions under user-specified constraints or Pareto trade-offs between different objectives (e.g., resource types and timing), which has not been achieved by existing HLS tools or DSE engines.
  - In Chapter 6, we design a DDPG-based method for workload placement optimization on MCMC systems [27], which takes into account the non-uniform and hierarchical on/off-chip communication capabilities. This method is scalable to large systems with thousands of cores and can handle different connection topologies without requiring topology-specific knowledge.

- **High-quality and productive design verification.** We use learning-based approaches to imitate conventional mechanisms so that modern computing power can

be more effectively utilized to promote verification productivity.

– In Chapter 7, we showcase the effectiveness of using the message-passing mechanism in GNN computation to emulate functional propagation and structural hashing in conventional symbolic reasoning methods [28], which achieves strong scalability to BNs with over 33 million nodes with up to six orders of magnitude speedups compared to the state-of-the-art implementation in the ABC framework.

## 8.2 Future Research

Going forward, we envision three enablers that should be jointly investigated to propel no-human-in-the-loop design automation: advanced algorithms, autonomous design methodology, and agile hardware development.

### 8.2.1 Advanced Algorithms: Specialized, Hierarchical, Explainable

Despite many victories of AI algorithms, the production-ready ML applications for hardware development await more endeavors to make algorithms specialized for target hardware problems. We will elaborate on several challenges that need to be addressed.

- *Handling Imperfect Data.* In hardware design or EDA tasks, the collected data are often imperfect for ML models, in terms of data scarcity, implicit labeling, and inevitable noise. In the case of data scarcity, generative models can generate synthetic data [282] that are artificial but realistic, which can be further incorporated with out-of-distribution methods [283] to improve generalization capability from synthetic to real data. In the case of lacking perfect labels, potential solutions are hy-

brid supervision techniques, such as self-supervised learning [284], semi-supervised learning [285], or combining supervised with unsupervised techniques [286]. In the case of noisy data, different data cleaning approaches [287, 288] are beneficial to improve model capability.

- *Multi-Level Abstraction for Performance Modeling.* Hardware development often adopts a bottom-up or top-down procedure, implying the importance of hierarchical structures for system-level characterization. Thus, advanced algorithms should leverage information from different levels of systems in synergy, including but not limited to multi-level abstraction and multi-modal representation, so that they can have better expressiveness on system behaviors.

- *Multi-Granularity Optimization.* EDA essentially is highly constrained multi-objective optimization. Many hardware system optimizations involve the participation of multiple components. These all speak to the necessity of multi-granularity optimization. Potential cures include hierarchical RL [289] that has flexible objective specifications and capabilities to learn goal-directed behaviors in complex environments with sparse feedback, and multi-agent RL [290] with fully cooperative, fully competitive, or mixed agents to facilitate versatile system optimization. Another thrust to improve optimization efficiency is to bridge continuous and discrete optimization, and our initial effort [291] using reparameterization tricks has shown its effectiveness in combinatorial optimization.

- *Explainable ML.* Interpretation and explanation regarding model behaviors are important to expose potential problems during training, encode expert knowledge into models, and ensure the fidelity of predictions. Thus, explainable ML techniques [292, 293] are expected to provide more confidence, reliability, and security for the decisions made in hardware design and verification.

### 8.2.2 Autonomous Hardware Design Methodology: Holistic, Heterogeneous, Scalable.

The scaling up and out of hardware systems feed into a much higher complexity of design automation, implying a rigid demand for a holistic and scalable design methodology that is capable to handle heterogeneity.

- *Holistic and Scalable Design and Optimization.* Next-generation EDA tools should target holistic design and provide system-wise compilation, synthesis, and optimization, which requires the development of a hierarchical and unified IR to represent heterogeneous hardware features and behaviors. MLIR [294] has demonstrated promise in compilation for heterogeneous hardware and facilitating optimizers at different levels of abstraction and across application domains and hardware targets. For instance, CIRCT [295], which applies MLIR [294] and LLVM [296] development methodology to the domain of hardware design tools, aims to build a reusable and consistent infrastructure and enable new higher-level abstractions for hardware design. By developing a reusable and extensible IR, future EDA tools can achieve scalability across various application scales, seamlessly integrate digital and analog circuits, and effectively combine traditional and emerging technologies.

- *Targeting System Heterogeneity.* Design and development of computer architectures often build upon earlier-generation architectures of similar purpose but incorporate next-generation hardware components that were unavailable in previous generations. In addition to the heterogeneity of components from different generations, modern hardware platforms typically involve heterogeneous processing units, such as CPU, GPU, FPGA, and ASIC. This heterogeneity is further compounded by increasingly complex workloads, such as multi-modal ML [297]. Thus, we antic-

ipate an intelligent design methodology capable of managing heterogeneity arising from hardware, workloads, and their mapping [298, 299, 300].

### 8.2.3 Agile Hardware Development: Portable, Reusable, Extensible.

Even though hardware specialization is essential in satisfying diverse design specifications (e.g., performance, robustness, power efficiency, environmental impact), non-recurring engineering (NRE) cost [301], development time [5, 264], and reconfigurability [302] remain challenging. Thus, the desired development flow should strike a balance between exploitation and exploration, i.e., integrating a generic step that promotes portability with an objective-specific step that guarantees performance. In addition, modular hardware blocks with well-defined interfaces will amortize NRE cost by higher reuse and can be easily extended to different workloads. One promising solution is chiplet-based hardware designs. A chiplet [303, 304] is a miniature IC with well-defined functionality and can be integrated with other chiplets using through silicon vias (TSVs) or interposers [305, 306]. Multiple chiplets can be combined using a mix-and-match approach, similar to assembling LEGO blocks, which significantly reduces the system-level design complexity. This also provides several advantages over a traditional system-on-chip (SoC): for example, IP blocks can be reused and assembled into many different hardware systems using existing and emerging integration technologies [307]; this modular approach also allows heterogeneous fabrication with different processes, materials, and technology nodes [305].

# Bibliography


[1] OpenAI, *Al and compute*, 2018. https://openai.com/blog/ai-and-compute/.

[2] G. E. Moore, *Cramming more components onto integrated circuits*, *Proceedings of the IEEE* **86** (1998), no. 1 82–85.

[3] C. A. Mack, *Fifty years of moore's law*, *IEEE Transactions on semiconductor manufacturing* **24** (2011), no. 2 202–207.

[4] M. M. Waldrop, *The chips are down for moore's law*, *Nature News* **530** (2016), no. 7589 144.

[5] H. Foster, *The 2020 wilson research group functional verification study*, 2020. https://blogs.sw.siemens.com/verificationhorizons/2020/10/27/prologue-the-2020-wilson-research-group-functional-verification-study/.

[6] F. Schirrmeister, P. Hardee, L. Melling, A. Dua, and M. Rubin, *Next generation verification for the era of ai/ml and 5g*, *Design and Verification Conference and Exhibition, US* (2020). https://dvcon-proceedings.org/document/next-generation-verification-for-the-era-of-ai-ml-and-5g/.

[7] M. Rosker, *Evolving the electronics resurgence initiative (eri 2.0)*, 2021. https://www.ndia.org/-/media/sites/ndia/divisions/electronics/eri2_ndia_20210421_releaseapproved_34584.ashx.

[8] N. Wu and Y. Xie, *A survey of machine learning for computer architecture and systems*, *ACM Computing Surveys* **55** (2022), no. 3 1–39.

[9] N. Wu, Y. Xie, and C. Hao, *Ai-assisted synthesis in next generation eda: Promises, challenges, and prospects*, in *Proceedings of the International Conference on Computer Design*, pp. 207–214, IEEE, 2022.

[10] Y. Chen, Y. Xie, L. Song, F. Chen, and T. Tang, *A survey of accelerator architectures for deep neural networks*, *Engineering* **6** (2020), no. 3 264–274.

[11] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, *Efficient processing of deep neural networks: A tutorial and survey*, *Proceedings of the IEEE* **105** (2017), no. 12 2295–2329.

[12] C. Mendis, A. Renda, S. Amarasinghe, and M. Carbin, *Ithemal: Accurate, portable and fast basic block throughput estimation using deep neural networks*, in *Proceedings of the International Conference on Machine Learning*, pp. 4505–4515, 2019.

[13] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, *et. al.*, *The gem5 simulator*, *ACM SIGARCH Computer Architecture News* **39** (2011), no. 2 1–7.

[14] T. E. Carlson, W. Heirman, and L. Eeckhout, *Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation*, in *Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–12, 2011.

[15] P. J. Van Laarhoven and E. H. Aarts, *Simulated annealing*, in *Simulated annealing: Theory and applications*, pp. 7–15. Springer, 1987.

[16] M. Mitchell, *An introduction to genetic algorithms*. MIT press, 1998.

[17] R. Eberhart and J. Kennedy, *Particle swarm optimization*, in *Proceedings of the IEEE International Conference on Neural Networks*, vol. 4, pp. 1942–1948, Citeseer, 1995.

[18] M. Dorigo, M. Birattari, and T. Stutzle, *Ant colony optimization*, *IEEE Computational Intelligence Magazine* **1** (2006), no. 4 28–39.

[19] E. Seligman, T. Schubert, and M. A. K. Kumar, *Formal verification: an essential toolkit for modern VLSI design*. Morgan Kaufmann Publishers Inc., 2015.

[20] H. D. Foster, *Trends in functional verification: A 2014 industry study*, in *Proceedings of the Design Automation Conference*, pp. 1–6, 2015.

[21] T. Bultiaux, S. Guenot, S. Hustin, A. Blampey, J. Bulone, and M. Moy, *Functional Verification*. Springer, 2005.

[22] L.-C. Wang, *Experience of data analytics in eda and test—principles, promises, and challenges*, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **36** (2016), no. 6 885–898.

[23] N. Wu, H. Yang, Y. Xie, P. Li, and C. Hao, *High-level synthesis performance prediction using gnns: Benchmarking, modeling, and advancing*, in *Proceedings of the Design Automation Conference*, pp. 1–6, IEEE, 2022.

[24] N. Wu, J. Lee, Y. Xie, and C. Hao, *Lostin: Logic optimization via spatio-temporal information with hybrid graph models*, in *Proceedings of the International Conference on Application-specific Systems, Architectures and Processors*, pp. 11–18, IEEE, 2022.

[25] N. Wu, Y. Xie, and C. Hao, *Ironman: Gnn-assisted design space exploration in high-level synthesis via reinforcement learning*, in *Proceedings of the Great Lakes Symposium on VLSI*, pp. 39–44, 2021.

[26] N. Wu, Y. Xie, and C. Hao, *Ironman-pro: Multi-objective design space exploration in hls via reinforcement learning and graph neural network based modeling*, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2022).

[27] N. Wu, L. Deng, G. Li, and Y. Xie, *Core placement optimization for multi-chip many-core neural network systems with reinforcement learning*, *ACM Transactions on Design Automation of Electronic Systems* **26** (2020), no. 2 1–27.

[28] N. Wu, Y. Li, C. Hao, S. Dai, C. Yu, and Y. Xie, *Gamora: Graph learning based symbolic reasoning for large-scale boolean networks*, in *Proceedings of the Design Automation Conference*, 2023.

[29] J. Shlomi, P. Battaglia, and J.-R. Vlimant, *Graph neural networks in particle physics*, *Machine Learning: Science and Technology* **2** (2020), no. 2 021001.

[30] X. Ju, S. Farrell, P. Calafiura, D. Murnane, L. Gray, T. Klijnsma, K. Pedro, G. Cerati, J. Kowalkowski, G. Perdue, *et. al.*, *Graph neural networks for particle reconstruction in high energy physics detectors*, *arXiv preprint arXiv:2003.11603* (2020).

[31] C. W. Coley, W. Jin, L. Rogers, T. F. Jamison, T. S. Jaakkola, W. H. Green, R. Barzilay, and K. F. Jensen, *A graph-convolutional neural network model for the prediction of chemical reactivity*, *Chemical science* **10** (2019), no. 2 370–377.

[32] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl, *Neural message passing for quantum chemistry*, in *Proceedings of the International Conference on Machine Learning*, pp. 1263–1272, PMLR, 2017.

[33] W. Fan, Y. Ma, Q. Li, Y. He, E. Zhao, J. Tang, and D. Yin, *Graph neural networks for social recommendation*, in *Proceedings of the World Wide Web Conference*, pp. 417–426, 2019.

[34] Z. Guo and H. Wang, *A deep graph neural network-based mechanism for social recommendations*, *IEEE Transactions on Industrial Informatics* **17** (2020), no. 4 2776–2783.

[35] T. Zhao, Y. Hu, L. R. Valsdottir, T. Zang, and J. Peng, *Identifying drug–target interactions based on graph convolutional network and deep neural network*, *Briefings in Bioinformatics* **22** (2021), no. 2 2141–2150.

[36] J. Lim, S. Ryu, K. Park, Y. J. Choe, J. Ham, and W. Y. Kim, *Predicting drug–target interaction using a novel graph neural network with 3d structure-embedded graph representation*, *Journal of Chemical Information and Modeling* **59** (2019), no. 9 3981–3988.

[37] J. Bruna, W. Zaremba, A. Szlam, and Y. LeCun, *Spectral networks and locally connected networks on graphs*, *arXiv preprint arXiv:1312.6203* (2013).

[38] H. Dai, B. Dai, and L. Song, *Discriminative embeddings of latent variable models for structured data*, in *Proceedings of the International Conference on Machine Learning*, pp. 2702–2711, PMLR, 2016.

[39] W. L. Hamilton, R. Ying, and J. Leskovec, *Inductive representation learning on large graphs*, in *Proceedings of the International Conference on Neural Information Processing Systems*, pp. 1025–1035, 2017.

[40] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, *et. al.*, *Mastering the game of go with deep neural networks and tree search*, *Nature* **529** (2016), no. 7587 484–489.

[41] B. R. Kiran, I. Sobh, V. Talpaert, P. Mannion, A. A. Al Sallab, S. Yogamani, and P. Pérez, *Deep reinforcement learning for autonomous driving: A survey*, *IEEE Transactions on Intelligent Transportation Systems* **23** (2021), no. 6 4909–4926.

[42] J. Kober, J. A. Bagnell, and J. Peters, *Reinforcement learning in robotics: A survey*, *The International Journal of Robotics Research* **32** (2013), no. 11 1238–1274.

[43] O. Vinyals, I. Babuschkin, W. M. Czarnecki, M. Mathieu, A. Dudzik, J. Chung, D. H. Choi, R. Powell, T. Ewalds, P. Georgiev, J. Oh, D. Horgan, M. Kroiss, I. Danihelka, A. Huang, L. Sifre, T. Cai, J. P. Agapiou, M. Jaderberg, A. S. Vezhnevets, R. Leblond, T. Pohlen, V. Dalibard, D. Budden, Y. Sulsky, J. Molloy, T. L. Paine, C. Gulcehre, Z. Wang, T. Pfaff, Y. Wu, R. Ring, D. Yogatama, D. Wünsch, K. McKinney, O. Smith, T. Schaul, T. Lillicrap, K. Kavukcuoglu, D. Hassabis, C. Apps, and D. Silver, *Grandmaster level in starcraft ii using multi-agent reinforcement learning*, *Nature* **575** (2019), no. 7782 350–354.

[44] J. Schrittwieser, I. Antonoglou, T. Hubert, K. Simonyan, L. Sifre, S. Schmitt, A. Guez, E. Lockhart, D. Hassabis, T. Graepel, *et. al.*, *Mastering atari, go, chess and shogi by planning with a learned model*, *Nature* **588** (2020), no. 7839 604–609.

[45] Y. Nevmyvaka, Y. Feng, and M. Kearns, *Reinforcement learning for optimized trade execution*, in *Proceedings of the International Conference on Machine Learning*, pp. 673–680, 2006.

[46] Y. Deng, F. Bao, Y. Kong, Z. Ren, and Q. Dai, *Deep direct reinforcement learning for financial signal representation and trading*, *IEEE Transactions on Neural Networks and Learning Systems* **28** (2016), no. 3 653–664.

[47] S. Sun, R. Wang, and B. An, *Reinforcement learning for quantitative trading*, *ACM Transactions on Intelligent Systems and Technology* **14** (2023), no. 3 1–29.

[48] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[49] C. J. Watkins and P. Dayan, *Q-learning*, *Machine Learning* **8** (1992), no. 3-4 279–292.

[50] R. J. Williams, *Simple statistical gradient-following algorithms for connectionist reinforcement learning*, *Machine learning* **8** (1992), no. 3-4 229–256.

[51] X. Dong, N. P. Jouppi, and Y. Xie, *A circuit-architecture co-optimization framework for exploring nonvolatile memory hierarchies*, *ACM Transactions on Architecture and Code Optimization* **10** (2013), no. 4 23.

[52] Z. Deng, L. Zhang, N. Mishra, H. Hoffmann, and F. T. Chong, *Memory cocktail therapy: a general learning-based framework to optimize dynamic tradeoffs in nvms*, in *Proceedings of the International Symposium on Microarchitecture*, pp. 232–244, ACM, 2017.

[53] D. Dai, F. S. Bao, J. Zhou, and Y. Chen, *Block2vec: A deep learning strategy on mining block correlations in storage systems*, in *Proceedings of the International Conference on Parallel Processing Workshops*, pp. 230–239, IEEE, 2016.

[54] Z. Shi, K. Swersky, D. Tarlow, P. Ranganathan, and M. Hashemi, *Learning execution through neural code fusion*, in *Proceedings of the International Conference on Learning Representations*, 2020.

[55] Z. Qian, D.-C. Juan, P. Bogdan, C.-Y. Tsui, D. Marculescu, and R. Marculescu, *Svr-noc: A performance analysis tool for network-on-chips using learning-based support vector regression model*, in *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition*, pp. 354–357, IEEE, 2013.

[56] V. Soteriou, T. Theocharides, and E. Kakoulli, *A holistic approach towards intelligent hotspot prevention in network-on-chip-based multicores*, *IEEE Transactions on Computers* **65** (2015), no. 3 819–833.

[57] M. Clark, A. Kodi, R. Bunescu, and A. Louri, *Lead: Learning-enabled energy-aware dynamic voltage/frequency scaling in nocs*, in *Proceedings of the Design Automation Conference*, p. 82, ACM, 2018.

[58] D. DiTomaso, T. Boraten, A. Kodi, and A. Louri, *Dynamic error mitigation in nocs using intelligent prediction techniques*, in *Proceedings of the International Symposium on Microarchitecture*, p. 31, IEEE, 2016.

[59] W. Jia, K. A. Shaw, and M. Martonosi, *Stargazer: Automated regression-based gpu design space exploration*, in *Proceedings of the International Symposium on Performance Analysis of Systems & Software*, pp. 2–13, IEEE, 2012.

[60] A. Pattnaik, X. Tang, A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, and C. R. Das, *Scheduling techniques for gpu architectures with processing-in-memory capabilities*, in *Proceedings of the International Conference on Parallel Architectures and Compilation*, pp. 31–44, ACM, 2016.

[61] G. Wu, J. L. Greathouse, A. Lyashevsky, N. Jayasena, and D. Chiou, *Gpgpu performance and power estimation using machine learning*, in *Proceedings of the International Symposium on High Performance Computer Architecture*, pp. 564–576, IEEE, 2015.

[62] K. O'Neal, P. Brisk, E. Shriver, and M. Kishinevsky, *Halwpe: Hardware-assisted light weight performance estimation for gpus*, in *Proceedings of the Design Automation Conference*, pp. 1–6, IEEE, 2017.

[63] E. İpek, S. A. McKee, R. Caruana, B. R. de Supinski, and M. Schulz, *Efficiently exploring architectural design spaces via predictive modeling*, *ACM SIGOPS Operating Systems Review* **40** (2006), no. 5 195–206.

[64] S. Khan, P. Xekalakis, J. Cavazos, and M. Cintra, *Using predictivemodeling for cross-program design space exploration in multicore systems*, in *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*, pp. 327–338, IEEE, 2007.

[65] B. C. Lee and D. M. Brooks, *Illustrative design space studies with microarchitectural regression models*, in *Proceedings of the International Symposium on High Performance Computer Architecture*, pp. 340–351, IEEE, 2007.

[66] B. C. Lee, J. Collins, H. Wang, and D. Brooks, *Cpr: Composable performance regression for scalable multiprocessor models*, in *Proceedings of the International Symposium on Microarchitecture*, pp. 270–281, IEEE, 2008.

[67] K. Sangaiah, M. Hempstead, and B. Taskin, *Uncore rpd: Rapid design space exploration of the uncore via regression modeling*, in *Proceedings of the International Conference on Computer-Aided Design*, pp. 365–372, IEEE, 2015.

[68] D. Lo, T. Song, and G. E. Suh, *Prediction-guided performance-energy trade-off for interactive applications*, in *Proceedings of the International Symposium on Microarchitecture*, pp. 508–520, ACM, 2015.

[69] N. Mishra, H. Zhang, J. D. Lafferty, and H. Hoffmann, *A probabilistic graphical model-based approach for minimizing energy under performance constraints*, *ACM SIGARCH Computer Architecture News* **43** (2015), no. 1 267–281.

[70] N. Mishra, C. Imes, J. D. Lafferty, and H. Hoffmann, *Caloree: Learning control for predictable latency and low energy*, in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 184–198, 2018.

[71] Y. Ding, N. Mishra, and H. Hoffmann, *Generative and multi-phase learning for computer systems optimization*, in *Proceedings of the International Symposium on Computer Architecture*, pp. 39–52, ACM, 2019.

[72] E. Teran, Z. Wang, and D. A. Jiménez, *Perceptron learning for reuse prediction*, in *Proceedings of the International Symposium on Microarchitecture*, pp. 1–12, IEEE, 2016.

[73] D. A. Jiménez and E. Teran, *Multiperspective reuse prediction*, in *Proceedings of the International Symposium on Microarchitecture*, pp. 436–448, IEEE, 2017.

[74] E. Bhatia, G. Chacon, S. Pugsley, E. Teran, P. V. Gratz, and D. A. Jiménez, *Perceptron-based prefetch filtering*, in *Proceedings of the International Symposium on Computer Architecture*, pp. 1–13, ACM, 2019.

[75] L. Peled, S. Mannor, U. Weiser, and Y. Etsion, *Semantic locality and context-based prefetching using reinforcement learning*, in *Proceedings of the International Symposium on Computer Architecture*, pp. 285–297, IEEE, 2015.

[76] N. Beckmann and D. Sanchez, *Maximizing cache performance under uncertainty*, in *Proceedings of the International Symposium on High Performance Computer Architecture*, pp. 109–120, IEEE, 2017.

[77] N. Wu and P. Li, *Phoebe: Reuse-aware online caching with reinforcement learning for emerging storage models*, *arXiv preprint arXiv:2011.07160* (2020).

[78] Y. Zeng and X. Guo, *Long short term memory based hardware prefetcher: A case study*, in *Proceedings of the International Symposium on Memory Systems*, pp. 305–311, ACM, 2017.

[79] M. Hashemi, K. Swersky, J. Smith, G. Ayers, H. Litz, J. Chang, C. Kozyrakis, and P. Ranganathan, *Learning memory access patterns*, in *Proceedings of the International Conference on Machine Learning*, pp. 1924–1933, 2018.

[80] Z. Shi, X. Huang, A. Jain, and C. Lin, *Applying deep learning to the cache replacement problem*, in *Proceedings of the International Symposium on Microarchitecture*, pp. 413–425, ACM, 2019.

[81] J. F. Martinez and E. Ipek, *Dynamic multicore resource management: A machine learning approach*, *IEEE micro* **29** (2009), no. 5 8–17.

[82] E. Ipek, O. Mutlu, J. F. Martínez, and R. Caruana, *Self-optimizing memory controllers: A reinforcement learning approach*, in *Proceedings of the International Symposium on Computer Architecture*, pp. 39–50, 2008.

[83] J. Mukundan and J. F. Martinez, *Morse: Multi-objective reconfigurable self-optimizing memory scheduler*, in *Proceedings of the International Symposium on High Performance Computer Architecture*, pp. 1–12, IEEE, 2012.

[84] D. A. Jiménez and C. Lin, *Dynamic branch prediction with perceptrons*, in *Proceedings of the International Symposium on High Performance Computer Architecture*, pp. 197–206, IEEE, 2001.

[85] D. A. Jiménez, *Fast path-based neural branch prediction*, in *Proceedings of the International Symposium on Microarchitecture*, pp. 243–252, IEEE, 2003.

[86] D. A. Jiménez, *Piecewise linear branch prediction*, in *Proceedings of the International Symposium on Computer Architecture*, pp. 382–393, IEEE, 2005.

[87] D. A. Jiménez, *Multiperspective perceptron predictor*, in *Proceedings of the 5th JILP Workshop on Computer Architecture Competitions (JWAC-5): Championship Branch Prediction*, 2016.

[88] E. Garza, S. Mirbagher-Ajorpaz, T. A. Khan, and D. A. Jiménez, *Bit-level perceptron prediction for indirect branches*, in *Proceedings of the International Symposium on Computer Architecture*, pp. 27–38, ACM, 2019.

[89] G. Zacharopoulos, A. Barbon, G. Ansaloni, and L. Pozzi, *Machine learning approach for loop unrolling factor prediction in high level synthesis*, in *Proceedings of the International Conference on High Performance Computing & Simulation*, pp. 91–97, IEEE, 2018.

[90] A. Mehrabi, A. Manocha, B. C. Lee, and D. J. Sorin, *Prospector: Synthesizing efficient accelerators via statistical learning*, in *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition*, pp. 151–156, IEEE, 2020.

[91] W. L. Neto, M. Austin, S. Temple, L. Amaru, X. Tang, and P.-E. Gaillardon, *Lsoracle: A logic synthesis framework driven by artificial intelligence*, in *Proceedings of the International Conference on Computer-Aided Design*, pp. 1–6, IEEE, 2019.

[92] W. Haaswijk, E. Collins, B. Seguin, M. Soeken, F. Kaplan, S. Süsstrunk, and G. De Micheli, *Deep learning for logic optimization algorithms*, in *Proceedings of the International Symposium on Circuits and Systems*, pp. 1–4, IEEE, 2018.

[93] G. S. Ravi and M. H. Lipasti, *Charstar: Clock hierarchy aware resource scaling in tiled architectures*, in *Proceedings of the International Symposium on Computer Architecture*, pp. 147–160, IEEE, 2017.

[94] C. Imes, S. Hofmeyr, and H. Hoffmann, *Energy-efficient application resource scheduling using machine learning classifiers*, in *Proceedings of the International Conference on Parallel Processing*, p. 45, ACM, 2018.

[95] G.-Y. Pan, J.-Y. Jou, and B.-C. Lai, *Scalable power management using multilevel reinforcement learning for multiprocessors*, *ACM Transactions on Design Automation of Electronic Systems* **19** (2014), no. 4 33.

[96] Z. Chen and D. Marculescu, *Distributed reinforcement learning for power limited many-core system performance optimization*, in *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition*, pp. 1521–1526, IEEE, 2015.

[97] Z. Chen, D. Stamoulis, and D. Marculescu, *Profit: priority and power/performance optimization for many-core systems*, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **37** (2017), no. 10 2064–2075.

[98] H. Hoffmann, *Jouleguard: energy guarantees for approximate applications*, in *Proceedings of the Symposium on Operating Systems Principles*, pp. 198–214, 2015.

[99] A. Piziali, *Functional verification coverage measurement and analysis*. Springer Science & Business Media, 2007.

[100] Y. Ma, H. Ren, B. Khailany, H. Sikka, L. Luo, K. Natarajan, and B. Yu, *High performance graph convolutional networks with applications in testability analysis*, in *Proceedings of the Design Automation Conference*, pp. 1–6, 2019.

[101] P. Krishnamurthy, A. B. Chowdhury, B. Tan, F. Khorrami, and R. Karri, *Explaining and interpreting machine learning cad decisions: An ic testing case study*, in *Proceedings of the 2nd Workshop on Machine Learning for CAD*, pp. 129–134, IEEE, 2020.

[102] V. Jayasree, *Machine learning for coverage analysis in design verification*, in *Proceedings of the Design and Verification Conference and Exhibition, Europe*, 2021.

[103] S. Sokorac, *Optimizing random test constraints using machine learning algorithms*, in *Proceedings of the Design and Verification Conference and Exhibition, US*, 2017.

[104] M. A. Abd El Ghany and K. A. Ismail, *Speed up functional coverage closure of cordic designs using machine learning models*, in *Proceedings of the International Conference on Microelectronics*, pp. 91–95, IEEE, 2021.

[105] R. Gal, E. Haber, B. Irwin, M. Mouallem, B. Saleh, and A. Ziv, *Using deep neural networks and derivative free optimization to accelerate coverage closure*, in *Proceedings of the 3rd Workshop on Machine Learning for CAD*, pp. 1–6, IEEE, 2021.

[106] S. Vasudevan, W. J. Jiang, D. Bieber, R. Singh, C. R. Ho, C. Sutton, *et. al.*, *Learning semantic representations to verify hardware designs*, *Proceedings of the International Conference on Neural Information Processing Systems* **34** (2021) 23491–23504.

[107] H. Choi, I. Huh, S. Kim, J. Ko, C. Jeong, H. Son, K. Kwon, J. Chai, Y. Park, J. Jeong, *et. al.*, *Application of deep reinforcement learning to dynamic verification of dram designs*, in *Proceedings of the Design Automation Conference*, pp. 523–528, IEEE, 2021.

[108] P.-H. Chang, D. Drmanac, and L.-C. Wang, *Online selection of effective functional test programs based on novelty detection*, in *Proceedings of the International Conference on Computer-Aided Design*, pp. 762–769, IEEE, 2010.

[109] W. Chen, N. Sumikawa, L.-C. Wang, J. Bhadra, X. Feng, and M. S. Abadir, *Novel test detection to improve simulation efficiency—a commercial experiment*, in *Proceedings of the International Conference on Computer-Aided Design*, pp. 101–108, IEEE, 2012.

[110] F. Wang, H. Zhu, P. Popli, Y. Xiao, P. Bodgan, and S. Nazarian, *Accelerating coverage directed test generation for functional verification: A neural network-based framework*, in *Proceedings of the Great Lakes Symposium on VLSI*, pp. 207–212, 2018.

[111] Q. Guo, T. Chen, Y. Chen, R. Wang, H. Chen, W. Hu, and G. Chen, *Pre-silicon bug forecast*, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **33** (2014), no. 3 451–463.

[112] A. Truong, D. Hellström, H. Duque, and L. Viklund, *Clustering and classification of uvm test failures using machine learning techniques*, in *Proceedings of the Design and Verification Conference and Exhibition, Europe*, 2018.

[113] E. E. Mandouh, L. Maher, M. Ahmed, Y. ElSharnoby, and A. G. Wassal, *Guiding functional verification regression analysis using machine learning and big data methods*, in *Proceedings of the Design and Verification Conference and Exhibition, Europe*, 2018.

[114] D. Maksimovic, A. Veneris, and Z. Poulos, *Clustering-based revision debug in regression verification*, in *Proceedings of the International Conference on Computer Design*, pp. 32–37, IEEE, 2015.

[115] B. Mammo, M. Furia, V. Bertacco, S. Mahlke, and D. S. Khudia, *Bugmd: Automatic mismatch diagnosis for bug triaging*, in *Proceedings of the International Conference on Computer-Aided Design*, pp. 1–7, IEEE, 2016.

[116] S. Dai, Y. Zhou, *et. al.*, *Fast and accurate estimation of quality of results in high-level synthesis with machine learning*, in *Proceedings of the International Symposium on Field-Programmable Custom Computing Machines*, pp. 129–132, IEEE, 2018.

[117] E. Ustun, C. Deng, *et. al.*, *Accurate operation delay prediction for fpga hls using graph neural networks*, in *Proceedings of the International Conference on Computer-Aided Design*, pp. 1–9, 2020.

[118] J. Zhao, L. Feng, S. Sinha, W. Zhang, Y. Liang, and B. He, *Comba: A comprehensive model-based analysis framework for high level synthesis of real applications*, in *Proceedings of the International Conference on Computer-Aided Design*, pp. 430–437, IEEE, 2017.

[119] J. Zhao, L. Feng, S. Sinha, W. Zhang, Y. Liang, and B. He, *Performance modeling and directives optimization for high-level synthesis on fpga*, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **39** (2019), no. 7 1428–1441.

[120] A. B. Perina, J. Becker, and V. Bonato, *Lina: Timing-constrained high-level synthesis performance estimator for fast dse*, in *Proceedings of the International Conference on Field-Programmable Technology*, pp. 343–346, IEEE, 2019.

[121] H. M. Makrani, F. Farahmand, H. Sayadi, S. Bondi, S. M. P. Dinakarrao, H. Homayoun, and S. Rafatirad, *Pyramid: Machine learning framework to estimate the optimal timing and resource usage of a high-level synthesis design*, in *Proceedings of the International Conference on Field Programmable Logic and Applications*, pp. 397–403, IEEE, 2019.

[122] H. M. Makrani, H. Sayadi, T. Mohsenin, S. Rafatirad, A. Sasan, and H. Homayoun, *Xppe: cross-platform performance estimation of hardware accelerators using machine learning*, in *Proceedings of the Asia and South Pacific Design Automation Conference*, pp. 727–732, 2019.

[123] K. O'Neal, M. Liu, H. Tang, A. Kalantar, K. DeRenard, and P. Brisk, *Hlspredict: Cross platform performance prediction for fpga high-level synthesis*, in *Proceedings of the International Conference on Computer-Aided Design*, pp. 1–8, 2018.

[124] H.-Y. Liu and L. P. Carloni, *On learning-based methods for design-space exploration with high-level synthesis*, in *Proceedings of the Design Automation Conference*, pp. 1–7, IEEE, 2013.

[125] P. Meng, A. Althoff, Q. Gautier, and R. Kastner, *Adaptive threshold non-pareto elimination: Re-thinking machine learning for system level design space exploration on fpgas*, in *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition*, pp. 918–923, IEEE, 2016.

[126] J. Kwon and L. P. Carloni, *Transfer learning for design-space exploration with high-level synthesis*, in *Proceedings of the 2nd Workshop on Machine Learning for CAD*, pp. 163–168, IEEE, 2020.

[127] D. Koeplinger, R. Prabhakar, Y. Zhang, C. Delimitrou, C. Kozyrakis, and K. Olukotun, *Automatic generation of efficient accelerators for reconfigurable hardware*, in *Proceedings of the International Symposium on Computer Architecture*, pp. 115–127, IEEE, 2016.

[128] J. Zhao, T. Liang, *et. al.*, *Machine learning based routing congestion prediction in fpga high-level synthesis*, in *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition*, pp. 1130–1135, IEEE, 2019.

[129] Z. Lin, J. Zhao, *et. al.*, *Hl-pow: A learning-based power modeling framework for high-level synthesis*, in *Proceedings of the Asia and South Pacific Design Automation Conference*, pp. 574–580, IEEE, 2020.

[130] A. Aho, M. Lam, R. Sethi, J. Ullman, K. Cooper, L. Torczon, and S. Muchnick, *Compilers: Principles, techniques and tools*, *Addison Wesley* (2007).

[131] M. Wolf, *Computers as components: principles of embedded computing system design*. Elsevier, 2012.

[132] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2011.

[133] G. Barany, *Liveness-driven random program generation*, in *Proceedings of the International Symposium on Logic-Based Program Synthesis and Transformation*, Springer, 2017.

[134] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, *Frama-c: A software analysis perspective*, *Formal Aspects of Computing* **27** (2015), no. 3 573–609.

[135] B. Reagen, R. Adolf, Y. S. Shao, G.-Y. Wei, and D. Brooks, *Machsuite: Benchmarks for accelerator design and customized architectures*, in *Proceedings of the International Symposium on Workload Characterization*, pp. 110–119, IEEE, 2014.

[136] Y. Hara, H. Tomiyama, S. Honda, and H. Takada, *Proposal and quantitative analysis of the chstone benchmark program suite for practical c-based high-level synthesis*, *Journal of Information Processing* **17** (2009) 242–254.

[137] L.-N. Pouchet and T. Yuki, "Polyhedral benchmark suite." `http://web.cs.ucla.edu/~pouchet/software/polybench/`, 2016.

[138] N. Wu, H. He, Y. Xie, P. Li, and C. Hao, *Program-to-circuit: Exploiting gnns for program representation and circuit translation*, *arXiv preprint arXiv:2109.06265* (2021).

[139] T. N. Kipf and M. Welling, *Semi-supervised classification with graph convolutional networks*, in *Proceedings of the International Conference on Learning Representations*, 2017.

[140] F. Wu, A. Souza, T. Zhang, C. Fifty, T. Yu, and K. Weinberger, *Simplifying graph convolutional networks*, in *Proceedings of the International Conference on Machine Learning*, pp. 6861–6871, PMLR, 2019.

[141] F. M. Bianchi, D. Grattarola, L. Livi, and C. Alippi, *Graph neural networks with convolutional arma filters*, *IEEE Transactions on Pattern Analysis and Machine Intelligence* **44** (2021), no. 7 3496–3507.

[142] Z. Ma, J. Xuan, Y. G. Wang, M. Li, and P. Liò, *Path integral based convolution and pooling for graph neural networks*, in *Proceedings of the International Conference on Neural Information Processing Systems*, pp. 16421–16433, 2020.

[143] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, *How powerful are graph neural networks?*, in *Proceedings of the International Conference on Learning Representations*, 2018.

[144] G. Corso, L. Cavalleri, D. Beaini, P. Liò, and P. Velickovic, *Principal neighbourhood aggregation for graph nets*, in *Proceedings of the International Conference on Neural Information Processing Systems*, pp. 13260–13271, 2020.

[145] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, *Graph attention networks*, in *Proceedings of the International Conference on Learning Representations*, 2018.

[146] Y. Li, R. Zemel, M. Brockschmidt, and D. Tarlow, *Gated graph sequence neural networks*, in *Proceedings of the International Conference on Learning Representations*, 2016.

[147] M. Schlichtkrull, T. N. Kipf, P. Bloem, R. van den Berg, I. Titov, and M. Welling, *Modeling relational data with graph convolutional networks*, in *Proceedings of the International Conference on Extended Semantic Web Conference*, pp. 593–607, Springer/Verlag, 2018.

[148] H. Gao and S. Ji, *Graph u-nets*, in *Proceedings of the International Conference on Machine Learning*, pp. 2083–2092, PMLR, 2019.

[149] M. Brockschmidt, *Gnn-film: Graph neural networks with feature-wise linear modulation*, in *Proceedings of the International Conference on Machine Learning*, pp. 1144–1152, PMLR, 2020.

[150] M. Fey and J. E. Lenssen, *Fast graph representation learning with pytorch geometric*, *arXiv preprint arXiv:1903.02428* (2019).

[151] Vitis, *Vitis High-Level Synthesis User Guide (UG1399)*. `https://docs.xilinx.com/r/en-US/ug1399-vitis-hls`.

[152] *Vivado Design Suite - HLx Editions*. `https://www.xilinx.com/products/design-tools/vivado.html`.

[153] H. Maron, H. Ben-Hamu, H. Serviansky, and Y. Lipman, *Provably powerful graph networks*, in *Proceedings of the International Conference on Neural Information Processing Systems*, pp. 2156–2167, 2019.

[154] *Spartan-3A FPGA Family: Data Sheet*. `https://www.xilinx.com/support/documentation/data_sheets/ds529.pdf`.

[155] J. Dyck, *Production-ready machine learning for eda*, 2018. `https://webinars.sw.siemens.com/production-ready-machine-learning/room`.

[156] C. Yu, H. Xiao, and G. De Micheli, *Developing synthesis flows without human knowledge*, in *Proceedings of the Design Automation Conference*, pp. 1–6, 2018.

[157] C. Yu and W. Zhou, *Decision making in synthesis cross technologies using lstms and transfer learning*, in *Proceedings of the Workshop on Machine Learning for CAD*, pp. 55–60, 2020.

[158] *Lynx design system*, Accessed: 2021. `https://www.synopsys.com/content/dam/synopsys/implementation&signoff/datasheets/lynx-design-system-ds.pdf`.

[159] R. Brayton and A. Mishchenko, *Abc: An academic industrial-strength verification tool*, in *Proceedings of the International Conference on Computer Aided Verification*, pp. 24–40, Springer, 2010.

[160] K. Zhu, M. Liu, H. Chen, Z. Zhao, and D. Z. Pan, *Exploring logic optimizations with reinforcement learning and graph convolutional network*, in *Proceedings of the Workshop on Machine Learning for CAD*, pp. 145–150, IEEE, 2020.

[161] A. Hosny, S. Hashemi, M. Shalan, and S. Reda, *Drills: Deep reinforcement learning for logic synthesis*, in *Proceedings of the Asia and South Pacific Design Automation Conference*, pp. 581–586, IEEE, 2020.

[162] B. Khailany, H. Ren, S. Dai, S. Godil, B. Keller, R. Kirby, A. Klinefelter, R. Venkatesan, Y. Zhang, B. Catanzaro, *et. al.*, *Accelerating chip design with machine learning*, *IEEE Micro* **40** (2020), no. 6 23–32.

[163] Y. Zhang, H. Ren, and B. Khailany, *Grannite: Graph neural network inference for transferable power estimation*, in *Proceedings of the Design Automation Conference*, pp. 1–6, IEEE, 2020.

[164] K. Ishiguro, S.-i. Maeda, and M. Koyama, *Graph warp module: an auxiliary module for boosting the power of graph neural networks in molecular graph analysis*, *arXiv preprint arXiv:1902.01020* (2019).

[165] S. Hochreiter and J. Schmidhuber, *Long short-term memory*, *Neural computation* **9** (1997), no. 8 1735–1780.

[166] D. Z. Pan, B. Halpin, and H. Ren, *Timing-driven placement*, *Handbook of Algorithms for Physical Design Automation* (2008) 423–446.

[167] A. B. Kahng, U. Mallappa, and L. Saul, *Using machine learning to predict path-based slack from graph-based timing analysis*, in *Proceedings of the International Conference on Computer Design*, pp. 603–612, IEEE, 2018.

[168] L. Amarú, P.-E. Gaillardon, and G. De Micheli, *Majority-inverter graph: A novel data-structure and algorithms for efficient logic optimization*, in *Proceedings of the Design Automation Conference*, pp. 1–6, IEEE, 2014.

[169] L. Amarú, P.-E. Gaillardon, and G. De Micheli, *The epfl combinational benchmark suite*, in *Proceedings of the International Workshop on Logic & Synthesis*, 2015.

[170] X. Xu, N. Shah, A. Evans, S. Sinha, B. Cline, and G. Yeric, *Standard cell library design and optimization methodology for asap7 pdk*, in *Proceedings of the International Conference on Computer-Aided Design*, pp. 999–1004, IEEE, 2017.

[171] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, *et. al.*, *Pytorch: An imperative style, high-performance deep learning library*, pp. 8026–8037, 2019.

[172] M. Liu, H. Gao, and S. Ji, *Towards deeper graph neural networks*, in *Proceedings of the International Conference on Knowledge Discovery & Data Mining*, pp. 338–348, 2020.

[173] M. H. Bateni, S. Behnezhad, M. Derakhshan, M. T. Hajiaghayi, R. Kiveris, S. Lattanzi, and V. Mirrokni, *Affinity clustering: hierarchical clustering at scale*, in *Proceedings of the International Conference on Neural Information Processing Systems*, pp. 6867–6877, 2017.

[174] A. Holzinger, B. Malle, A. Saranti, and B. Pfeifer, *Towards multi-modal causability with graph neural networks enabling information fusion for explainable ai*, *Information Fusion* **71** (2021) 28–37.

[175] Y. Wei, X. Wang, L. Nie, X. He, R. Hong, and T.-S. Chua, *Mmgcn: Multi-modal graph convolution network for personalized recommendation of micro-video*, in *Proceedings of the International Conference on Multimedia*, pp. 1437–1445, 2019.

[176] T. Tong, K. Gray, Q. Gao, L. Chen, D. Rueckert, A. D. N. Initiative, *et. al.*, *Multi-modal classification of alzheimer's disease using nonlinear graph fusion*, *Pattern recognition* **63** (2017) 171–181.

[177] *Design compiler*, Accessed: 2022. `https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/dc-ultra.html`.

[178] *Genus synthesis solution*, Accessed: 2022. `https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/synthesis/genus-synthesis-solution.html`.

[179] Z. Hu, T. Chen, K.-W. Chang, and Y. Sun, *Few-shot representation learning for out-of-vocabulary words*, in *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, 2019.

[180] S. Ruder, M. E. Peters, S. Swayamdipta, and T. Wolf, *Transfer learning in natural language processing*, in *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics: Tutorials*, pp. 15–18, 2019.

[181] *Xilinx Vivado High-Level Synthesis*. `https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html`.

[182] *Cadance Stratus High-Level Synthesis*. `https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/synthesis/stratus-high-level-synthesis.html`.

[183] J. Cong, P. Zhang, and Y. Zou, *Optimizing memory hierarchy allocation with loop transformations for high-level synthesis*, in *Proceedings of the Design Automation Conference*, pp. 1233–1238, 2012.

[184] W. Zuo, Y. Liang, P. Li, K. Rupnow, D. Chen, and J. Cong, *Improving high level synthesis optimization opportunity through polyhedral transformations*, in *Proceedings of the International Symposium on Field Programmable Gate Arrays*, pp. 9–18, 2013.

[185] B. C. Schafer and Z. Wang, *High-level synthesis design space exploration: Past, present, and future*, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **39** (2019), no. 10 2628–2639.

[186] J. d. F. Licht, M. Besta, S. Meierhans, and T. Hoefler, *Transformations of high-level synthesis codes for high-performance computing*, *IEEE Transactions on Parallel and Distributed Systems* **32** (2020), no. 5 1014–1029.

[187] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski, *Legup: high-level synthesis for fpga-based processor/accelerator systems*, in *Proceedings of the International Symposium on Field Programmable Gate Arrays*, pp. 33–36, 2011.

[188] R. Nane, V.-M. Sima, B. Olivier, R. Meeuws, Y. Yankova, and K. Bertels, *Dwarv 2.0: A cosy-based c-to-vhdl hardware compiler*, in *Proceedings of the International Conference on Field Programmable Logic and Applications*, pp. 619–622, IEEE, 2012.

[189] C. Pilato and F. Ferrandi, *Bambu: A modular framework for the high level synthesis of memory-intensive applications*, in *Proceedings of the International Conference on Field programmable Logic and Applications*, pp. 1–4, IEEE, 2013.

[190] *Intel High Level Synthesis Compiler Pro Edition*. `https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/hls/mnl-hls-reference.pdf`.

[191] G. Zhong, A. Prakash, Y. Liang, T. Mitra, and S. Niar, *Lin-analyzer: A high-level performance analysis tool for fpga-based accelerators*, in *Proceedings of the Design Automation Conference*, pp. 1–6, IEEE, 2016.

[192] D. Liu and B. C. Schafer, *Efficient and reliable high-level synthesis design space explorer for fpgas*, in *Proceedings of the International Conference on Field Programmable Logic and Applications*, pp. 1–8, IEEE, 2016.

[193] Z. Wang and B. C. Schafer, *Machine learning to set meta-heuristic specific parameters for high-level synthesis design space exploration*, in *Proceedings of the Design Automation Conference*, pp. 1–6, IEEE, 2020.

[194] A. Mahapatra and B. C. Schafer, *Machine-learning based simulated annealer method for high level synthesis design space exploration*, in *Proceedings of the Electronic System Level Synthesis Conference*, pp. 1–6, IEEE, 2014.

[195] R. G. Kim, J. R. Doppa, and P. P. Pande, *Machine learning for design space exploration and optimization of manycore systems*, in *Proceedings of the International Conference on Computer-Aided Design*, pp. 1–6, IEEE, 2018.

[196] D. H. Ram, M. Bhuvaneswari, and S. Logesh, *A novel evolutionary technique for multi-objective power, area and delay optimization in high level synthesis of datapaths*, in *Proceedings of the IEEE Computer Society Annual Symposium on VLSI*, pp. 290–295, 2011.

[197] B. C. Schafer, *Parallel high-level synthesis design space exploration for behavioral ips of exact latencies*, *ACM Transactions on Design Automation of Electronic Systems* **22** (2017), no. 4 1–20.

[198] B. C. Schafer, T. Takenaka, and K. Wakabayashi, *Adaptive simulated annealer for high level synthesis design space exploration*, in *Proceedings of the International Symposium on VLSI Design, Automation and Test*, pp. 106–109, IEEE, 2009.

[199] Y. Zhang, S. Wang, and G. Ji, *A comprehensive survey on particle swarm optimization algorithm and its applications*, *Mathematical Problems in Engineering* **2015** (2015).

[200] M. Zuluaga, A. Krause, G. Sergent, and M. Püschel, *Active learning for multi-objective optimization*, in *Proceedings of the 30th International Conference on International Conference on Machine Learning*, pp. 462–470, PMLR, 2013.

[201] Q. Sun, T. Chen, S. Liu, J. Chen, H. Yu, and B. Yu, *Correlated multi-objective multi-fidelity optimization for hls directives design*, *ACM Transactions on Design Automation of Electronic Systems* **27** (2022), no. 4 1–27.

[202] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, *The graph neural network model*, *IEEE Transactions on Neural Networks* **20** (2008), no. 1 61–80.

[203] *Implementing Multipliers in FPGA Devices*. `https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/an/an306.pdf`.

[204] *Spartan-3A FPGA Family: Data Sheet*. `https://www.xilinx.com/support/documentation/data_sheets/ds529.pdf`.

[205] *Zynq-7000 SoC (Z-7007S, Z-7012S, Z-7014S, Z-7010, Z-7015, and Z-7020): DC and AC Switching Characteristics*. `https://www.xilinx.com/support/documentation/data_sheets/ds187-XC7Z010-XC7Z020-Data-Sheet.pdf`.

[206] S.-C. Kao, G. Jeong, and T. Krishna, *Confuciux: Autonomous hardware resource assignment for dnn accelerators using reinforcement learning*, in *Proceedings of the International Symposium on Microarchitecture*, pp. 622–636, IEEE, 2020.

[207] R. S. Sutton, D. McAllester, S. Singh, and Y. Mansour, *Policy gradient methods for reinforcement learning with function approximation*, in *Proceedings of the International Conference on Neural Information Processing Systems*, pp. 1057–1063, 1999.

[208] D. Whitley, *A genetic algorithm tutorial*, *Statistics and Computing* **4** (1994), no. 2 65–85.

[209] J. Panerati, D. Sciuto, and G. Beltrame, *Optimization strategies in design space exploration*, in *Handbook of Hardware/Software Codesign*, pp. 189–216. Springer, 2017.

[210] K. Shilpa, C. LakshmiNarayana, and M. K. Singh, *Optimal resource allocation and binding in high-level synthesis using nature-inspired computation*, in *Emerging Research in Electronics, Computer Science and Technology*, pp. 1107–1118. Springer, 2019.

[211] Q. Li, Z. Han, and X.-M. Wu, *Deeper insights into graph convolutional networks for semi-supervised learning*, in *Proceedings of the AAAI conference on artificial intelligence*, vol. 32, 2018.

[212] B. C. Schafer, *Probabilistic multiknob high-level synthesis design space exploration acceleration*, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **35** (2015), no. 3 394–406.

[213] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, *Shidiannao: Shifting vision processing closer to the sensor*, in *Proceedings of the International Symposium on Computer Architecture*, pp. 92–104, 2015.

[214] Y.-H. Chen, J. Emer, and V. Sze, *Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks*, .

[215] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, *Scnn: An accelerator for compressed-sparse convolutional neural networks*, in *Proceedings of the International Symposium on Computer Architecture*, pp. 27–40, IEEE, 2017.

[216] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam, *Dadiannao: A machine-learning supercomputer*, in *Proceedings of the International Symposium on Microarchitecture*, pp. 609–622, IEEE, 2014.

[217] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen, *Cambricon-x: An accelerator for sparse neural networks*, in *Proceedings of the International Symposium on Microarchitecture*, pp. 1–12, IEEE, 2016.

[218] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, *Eie: efficient inference engine on compressed deep neural network*, in *Proceedings of the International Symposium on Computer Architecture*, pp. 243–254, IEEE, 2016.

[219] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, *In-datacenter performance analysis of a tensor processing unit*, in *Proceedings of the International Symposium on Computer Architecture*, pp. 1–12, IEEE, 2017.

[220] D. Shin, J. Lee, J. Lee, and H.-J. Yoo, *14.2 dnpu: An 8.1 tops/w reconfigurable cnn-rnn processor for general-purpose deep neural networks*, in *Proceedings of the International Solid-State Circuits Conference*, pp. 240–241, IEEE, 2017.

[221] E. Painkras, L. A. Plana, J. Garside, S. Temple, F. Galluppi, C. Patterson, D. R. Lester, A. D. Brown, and S. B. Furber, *Spinnaker: A 1-w 18-core system-on-chip for massively-parallel neural network simulation*, *IEEE Journal of Solid-State Circuits* **48** (2013), no. 8 1943–1953.

[222] F. Akopyan, J. Sawada, A. Cassidy, R. Alvarez-Icaza, J. Arthur, P. Merolla, N. Imam, Y. Nakamura, P. Datta, G.-J. Nam, B. Taba, M. Beakes, B. Brezzo, J. B. Kuang, R. Manohar, W. P. Risk, B. Jackson, and D. S. Modha, *Truenorth: Design and tool flow of a 65 mw 1 million neuron programmable neurosynaptic chip*, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **34** (2015), no. 10 1537–1557.

[223] M. Davies, N. Srinivasa, T.-H. Lin, G. Chinya, Y. Cao, S. H. Choday, G. Dimou, P. Joshi, N. Imam, S. Jain, Y. Liao, C.-K. Lin, A. Lines, R. Liu, D. Mathaikutty, S. McCoy, A. Paul, J. Tse, G. Venkataramanan, Y.-H. Weng, A. Wild, Y. Yang, and H. Wang, *Loihi: A neuromorphic manycore processor with on-chip learning*, *IEEE Micro* **38** (2018), no. 1 82–99.

[224] J. Pei, L. Deng, S. Song, M. Zhao, Y. Zhang, S. Wu, G. Wang, Z. Zou, Z. Wu, W. He, F. Chen, N. Deng, S. Wu, Y. Wang, Y. Wu, Z. Yang, C. Ma, G. Li, W. Han, H. Li, H. Wu, R. Zhao, Y. Xie, and L. Shi, *Towards artificial general*

*intelligence with hybrid tianjic chip architecture*, *Nature* **572** (2019), no. 7767 106–111.

[225] L. Deng, G. Wang, G. Li, S. Li, L. Liang, M. Zhu, Y. Wu, Z. Yang, Z. Zou, J. Pei, *et. al.*, *Tianjic: A unified and scalable chip bridging spike-based and continuous neural computation*, *IEEE Journal of Solid-State Circuits* **55** (2020), no. 8 2228–2246.

[226] Y. S. Shao, J. Clemons, R. Venkatesan, B. Zimmer, M. Fojtik, N. Jiang, B. Keller, A. Klinefelter, N. Pinckney, P. Raina, S. G. Tell, Y. Zhang, W. J. Dally, J. Emer, C. T. Gray, B. Khailany, and S. W. Keckler, *Simba: Scaling deep-learning inference with multi-chip-module-based architecture*, in *Proceedings of the International Symposium on Microarchitecture*, pp. 14–27, 2019.

[227] A. Ankit, I. E. Hajj, S. R. Chalamalasetti, G. Ndu, M. Foltin, R. S. Williams, P. Faraboschi, W.-m. W. Hwu, J. P. Strachan, K. Roy, *et. al.*, *Puma: A programmable ultra-efficient memristor-based accelerator for machine learning inference*, in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 715–731, 2019.

[228] S. K. Esser, P. A. Merolla, J. V. Arthur, A. S. Cassidy, R. Appuswamy, A. Andreopoulos, D. J. Berg, J. L. McKinstry, T. Melano, D. R. Barch, C. di Nolfo, P. Datta, A. Amir, B. Taba, M. D. Flickner, and D. S. Modha, *Convolutional networks for fast energy-efficient neuromorphic computing*, *Proceedings of the National Academy of Sciences* **113** (2016), no. 41 11441–11446.

[229] G. Urgese, F. Barchi, E. Macii, and A. Acquaviva, *Optimizing network traffic for spiking neural network simulations on densely interconnected many-core neuromorphic platforms*, *IEEE Transactions on Emerging Topics in Computing* **6** (2016), no. 3 317–329.

[230] M. K. F. Lee, Y. Cui, T. Somu, T. Luo, J. Zhou, W. T. Tang, W.-F. Wong, and R. S. M. Goh, *A system-level simulator for rram-based neuromorphic computing chips*, *ACM Transactions on Architecture and Code Optimization* **15** (2019), no. 4 1–24.

[231] L. Song, J. Mao, Y. Zhuo, X. Qian, H. Li, and Y. Chen, *Hypar: Towards hybrid parallelism for deep learning accelerator array*, in *Proceedings of the International Symposium on High Performance Computer Architecture*, pp. 56–68, IEEE, 2019.

[232] J. Hu and R. Marculescu, *Energy-aware mapping for tile-based noc architectures under performance constraints*, in *Proceedings of the Asia and South Pacific Design Automation Conference*, pp. 233–239, 2003.

[233] J. Hu and R. Marculescu, *Energy-and performance-aware mapping for regular noc architectures*, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **24** (2005), no. 4 551–562.

[234] P. K. Sharma, S. Biswas, and P. Mitra, *Energy efficient heuristic application mapping for 2-d mesh-based network-on-chip*, *Microprocessors and Microsystems* **64** (2019) 88–100.

[235] W.-T. Shen, C.-H. Chao, Y.-K. Lien, and A.-Y. Wu, *A new binomial mapping and optimization algorithm for reduced-complexity mesh-based on-chip network*, in *Proceedings of the International Symposium on Networks-on-Chip*, pp. 317–322, IEEE, 2007.

[236] T. Lei and S. Kumar, *A two-step genetic algorithm for mapping task graphs to a network on chip architecture*, in *Proceedings of the Euromicro Symposium on Digital System Design*, pp. 180–187, IEEE, 2003.

[237] S. Murali and G. De Micheli, *Bandwidth-constrained mapping of cores onto noc architectures*, in *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition*, vol. 3, pp. 20896–20896, IEEE, 2004.

[238] P. K. Sahu, N. Shah, K. Manna, and S. Chattopadhyay, *A new application mapping algorithm for mesh based network-on-chip design*, in *Proceedings of the India Conference*, pp. 1–4, IEEE, 2010.

[239] M. R. Garey and D. S. Johnson, *Computers and intractability*, vol. 174. Freeman San Francisco, 1979.

[240] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller, *Deterministic policy gradient algorithms*, in *Proceedings of the International Conference on Machine Learning*, pp. 387–395, PMLR, 2014.

[241] A. Krizhevsky, I. Sutskever, and G. E. Hinton, *Imagenet classification with deep convolutional neural networks*, in *Proceedings of the International Conference on Neural Information Processing Systems*, pp. 1097–1105, 2012.

[242] K. Simonyan and A. Zisserman, *Very deep convolutional networks for large-scale image recognition*, *arXiv preprint arXiv:1409.1556* (2014).

[243] K. He, X. Zhang, S. Ren, and J. Sun, *Deep residual learning for image recognition*, in *Proceedings of the Conference on Computer Vision and Pattern Recognition*, pp. 770–778, 2016.

[244] A. Boni, A. Pierazzi, and D. Vecchi, *Lvds i/o interface for gb/s-per-pin operation in 0.35-/spl mu/m cmos*, *IEEE Journal of Solid-State Circuits* **36** (2001), no. 4 706–711.

[245] T. Beukema, M. Sorna, K. Selander, S. Zier, B. L. Ji, P. Murfet, J. Mason, W. Rhee, H. Ainspan, B. Parker, and M. Beakes, *A 6.4-gb/s cmos serdes core with feed-forward and decision-feedback equalization*, *IEEE Journal of Solid-State Circuits* **40** (2005), no. 12 2633–2645.

[246] J. M. Wilson, W. J. Turner, J. W. Poulton, B. Zimmer, X. Chen, S. S. Kudva, S. Song, S. G. Tell, N. Nedovic, W. Zhao, S. R. Sudhakaran, C. T. Gray, and W. J. Dally, *A 1.17 pj/b 25gb/s/pin ground-referenced single-ended serial link for off-and on-package communication in 16nm cmos using a process-and temperature-adaptive voltage regulator*, in *Proceedings of the International Solid-State Circuits Conference*, pp. 276–278, IEEE, 2018.

[247] B. Zimmer, R. Venkatesan, Y. S. Shao, J. Clemons, M. Fojtik, N. Jiang, B. Keller, A. Klinefelter, N. Pinckney, P. Raina, *et. al.*, *A 0.11 pj/op, 0.32-128 tops, scalable multi-chip-module-based deep neural network accelerator with ground-reference signaling in 16nm*, in *Proceedings of the Symposium on VLSI Circuits*, pp. C300–C301, IEEE, 2019.

[248] A. Mirhoseini, H. Pham, Q. V. Le, B. Steiner, R. Larsen, Y. Zhou, N. Kumar, M. Norouzi, S. Bengio, and J. Dean, *Device placement optimization with reinforcement learning*, in *Proceedings of the International Conference on Machine Learning*, pp. 2430–2439, JMLR, 2017.

[249] A. Mirhoseini, A. Goldie, H. Pham, B. Steiner, Q. V. Le, and J. Dean, *A hierarchical model for device placement*, in *Proceedings of the International Conference on Learning Representations*, 2018.

[250] Y. Gao, L. Chen, and B. Li, *Spotlight: Optimizing device placement for training deep neural networks*, in *Proceedings of the International Conference on Machine Learning*, pp. 1676–1684, PMLR, 2018.

[251] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, *Proximal policy optimization algorithms*, *arXiv preprint arXiv:1707.06347* (2017).

[252] Y. Gao, L. Chen, and B. Li, *Post: device placement with cross-entropy minimization and proximal policy optimization*, in *Proceedings of the International Conference on Neural Information Processing Systems*, pp. 9993–10002, 2018.

[253] R. Addanki, S. B. Venkatakrishnan, S. Gupta, H. Mao, and M. Alizadeh, *Placeto: Efficient progressive device placement optimization*, in *Machine Learning for Systems Workshop, collocated with the International Conference on Neural Information Processing Systems*, 2018.

[254] X. Yang, M. Gao, J. Pu, A. Nayak, Q. Liu, S. E. Bell, J. O. Setter, K. Cao, H. Ha, C. Kozyrakis, and M. Horowitz, *Dnn dataflow choice is overrated*, *arXiv preprint arXiv:1809.04070* (2018).

[255] L. Deng, L. Liang, G. Wang, L. Chang, X. Hu, X. Ma, L. Liu, J. Pei, G. Li, and Y. Xie, *Semimap: A semi-folded convolution mapping for speed-overhead balance on crossbars*, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **39** (2018), no. 1 117–130.

[256] Y. LeCun, C. Cortes, and C. J. Burges, *Mnist handwritten digit database*, 2010. `http://yann.lecun.com/exdb/mnist`.

[257] G. E. Uhlenbeck and L. S. Ornstein, *On the theory of the brownian motion*, *Physical Review* **36** (1930), no. 5 823.

[258] S. Carrillo, J. Harkin, L. J. McDaid, F. Morgan, S. Pande, S. Cawley, and B. McGinley, *Scalable hierarchical network-on-chip architecture for spiking neural network hardware implementations*, *IEEE Transactions on Parallel and Distributed Systems* **24** (2012), no. 12 2451–2461.

[259] J. Kim, W. J. Dally, S. Scott, and D. Abts, *Technology-driven, highly-scalable dragonfly topology*, in *Proceedings of the International Symposium on Computer Architecture*, pp. 77–88, IEEE, 2008.

[260] M. Ciesielski, T. Su, A. Yasin, and C. Yu, *Understanding algebraic rewriting for arithmetic circuit verification: a bit-flow model*, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **39** (2019), no. 6 1346–1357.

[261] A. Mahzoon, D. Große, and R. Drechsler, *Revsca: Using reverse engineering to bring light into backward rewriting for big and dirty multipliers*, in *Proceedings of the Design Automation Conference*, pp. 1–6, 2019.

[262] H. Li, S. Patnaik, A. Sengupta, H. Yang, J. Knechtel, B. Yu, E. F. Young, and O. Sinanoglu, *Attacking split manufacturing from a deep learning perspective*, in *Proceedings of the Design Automation Conference*, pp. 1–6, 2019.

[263] U. J. Botero, R. Wilson, H. Lu, M. T. Rahman, M. A. Mallaiyan, F. Ganji, N. Asadizanjani, M. M. Tehranipoor, D. L. Woodard, and D. Forte, *Hardware trust and assurance through reverse engineering: A tutorial and outlook from image analysis and machine learning perspectives*, *ACM Journal on Emerging Technologies in Computing Systems* **17** (2021), no. 4 1–53.

[264] H. Foster, *The 2022 wilson research group functional verification study*, 2022. `https://blogs.sw.siemens.com/verificationhorizons/2022/10/10/prologue-the-2022-wilson-research-group-functional-verification-study/`.

[265] W. Li, A. Gascon, P. Subramanyan, W. Y. Tan, A. Tiwari, S. Malik, N. Shankar, and S. A. Seshia, *Wordrev: Finding word-level structures in a sea of bit-level*

*gates*, in *Proceedings of the International Symposium on Hardware-Oriented Security and Trust*, pp. 67–74, IEEE, 2013.

[266] B. Cakir and S. Malik, *Reverse engineering digital ics through geometric embedding of circuit graphs*, *ACM Transactions on Design Automation of Electronic Systems* **23** (2018), no. 4 1–19.

[267] P. Subramanyan, N. Tsiskaridze, W. Li, A. Gascón, W. Y. Tan, A. Tiwari, N. Shankar, S. A. Seshia, and S. Malik, *Reverse engineering digital circuits using structural and functional analyses*, *IEEE Transactions on Emerging Topics in Computing* **2** (2013), no. 1 63–80.

[268] A. Gascón, P. Subramanyan, B. Dutertre, A. Tiwari, D. Jovanović, and S. Malik, *Template-based circuit understanding*, in *Proceedings of the Conference on Formal Methods in Computer-Aided Design*, pp. 83–90, IEEE, 2014.

[269] C. Yu, M. Ciesielski, and A. Mishchenko, *Fast algebraic rewriting based on and-inverter graphs*, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **37** (2017), no. 9 1907–1911.

[270] A. Mishchenko, S. Chatterjee, and R. Brayton, *Dag-aware aig rewriting a fresh look at combinational logic synthesis*, in *Proceedings of the Design Automation Conference*, pp. 532–535, 2006.

[271] L. Alrahis, A. Sengupta, J. Knechtel, S. Patnaik, H. Saleh, B. Mohammad, M. Al-Qutayri, and O. Sinanoglu, *Gnn-re: Graph neural networks for reverse engineering of gate-level netlists*, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **41** (2021), no. 8 2435–2448.

[272] T. Bücher, L. Alrahis, G. Paim, S. Bampi, O. Sinanoglu, and H. Amrouch, *Appgnn: Approximation-aware functional reverse engineering using graph neural networks*, in *Proceedings of the International Conference on Computer-Aided Design*, pp. 1–9, 2022.

[273] G. Zhao and K. Shamsi, *Graph neural network based netlist operator detection under circuit rewriting*, in *Proceedings of the Great Lakes Symposium on VLSI*, pp. 53–58, 2022.

[274] Z. He, Z. Wang, C. Bail, H. Yang, and B. Yu, *Graph learning-based arithmetic block identification*, in *Proceedings the International Conference On Computer-Aided Design*, pp. 1–8, IEEE, 2021.

[275] A. Acar, H. Aksu, A. S. Uluagac, and M. Conti, *A survey on homomorphic encryption schemes: Theory and implementation*, *ACM Computing Surveys* **51** (2018), no. 4 1–35.

[276] M. Temel and W. A. Hunt, *Sound and automated verification of real-world rtl multipliers*, in *Proceedings of the Conference on Formal Methods in Computer Aided Design*, pp. 53–62, IEEE, 2021.

[277] D. Kaufmann, A. Biere, and M. Kauers, *Verifying large multipliers by combining sat and computer algebra*, in *Proceedings of the Conference on Formal Methods in Computer Aided Design*, pp. 28–36, IEEE, 2019.

[278] A. Mahzoon, D. Große, C. Scholl, A. Konrad, and R. Drechsler, *Formal verification of modular multipliers using symbolic computer algebra and boolean satisfiability*, in *Proceedings of the Design Automation Conference*, pp. 1183–1188, 2022.

[279] E. M. Sentovich, *Sis: A system for sequential circuit synthesis*, *Memorandom no. UCB/ERL M92/41* (1992).

[280] C. Yu, W. Brown, D. Liu, A. Rossi, and M. Ciesielski, *Formal verification of arithmetic circuits by function extraction*, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **35** (2016), no. 12 2131–2142.

[281] V. Vapnik, *Principles of risk minimization for learning theory*, in *Proceedings of the International Conference on Neural Information Processing Systems*, pp. 831–838, 1991.

[282] A. Alaa, B. Van Breugel, E. S. Saveliev, and M. van der Schaar, *How faithful is your synthetic data? sample-level metrics for evaluating and auditing generative models*, in *Proceedings of the International Conference on Machine Learning*, pp. 290–306, PMLR, 2022.

[283] H. Li, X. Wang, Z. Zhang, and W. Zhu, *Ood-gnn: Out-of-distribution generalized graph neural network*, *IEEE Transactions on Knowledge & Data Engineering* (2022), no. 01 1–14.

[284] D. Hendrycks, M. Mazeika, S. Kadavath, and D. Song, *Using self-supervised learning can improve model robustness and uncertainty*, *Proceedings of the International Conference on Neural Information Processing Systems* **32** (2019).

[285] J. E. Van Engelen and H. H. Hoos, *A survey on semi-supervised learning*, *Machine Learning* **109** (2020), no. 2 373–440.

[286] M. Alawieh, F. Wang, and X. Li, *Efficient hierarchical performance modeling for integrated circuits via bayesian co-learning*, in *Proceedings of the Design Automation Conference*, pp. 1–6, 2017.

[287] I. F. Ilyas and X. Chu, *Data cleaning*. Morgan & Claypool, 2019.

[288] S. Gupta and A. Gupta, *Dealing with noise problem in machine learning data-sets: A systematic review*, *Procedia Computer Science* **161** (2019) 466–474.

[289] T. D. Kulkarni, K. Narasimhan, A. Saeedi, and J. Tenenbaum, *Hierarchical deep reinforcement learning: Integrating temporal abstraction and intrinsic motivation*, *Proceedings of the International Conference on Neural Information Processing Systems* **29** (2016).

[290] K. Zhang, Z. Yang, and T. Başar, *Multi-agent reinforcement learning: A selective overview of theories and algorithms*, *Handbook of Reinforcement Learning and Control* (2021) 321–384.

[291] H. Wang, N. Wu, H. Yang, C. Hao, and P. Li, *Unsupervised learning for combinatorial optimization with principled objective relaxation*, in *Proceedings of the International Conference on Neural Information Processing Systems*, 2022.

[292] L. H. Gilpin, D. Bau, B. Z. Yuan, A. Bajwa, M. Specter, and L. Kagal, *Explaining explanations: An overview of interpretability of machine learning*, in *Proceedings of the International Conference on Data Science and Advanced Analytics*, pp. 80–89, IEEE, 2018.

[293] D. V. Carvalho, E. M. Pereira, and J. S. Cardoso, *Machine learning interpretability: A survey on methods and metrics*, *Electronics* **8** (2019), no. 8 832.

[294] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, *Mlir: Scaling compiler infrastructure for domain specific computation*, in *Proceedings of the International Symposium on Code Generation and Optimization*, pp. 2–14, IEEE, 2021.

[295] CIRCT, "Circuit ir compilers and tools." `https://circt.llvm.org/`.

[296] C. Lattner and V. Adve, *Llvm: A compilation framework for lifelong program analysis & transformation*, in *Proceedings of the International Symposium on Code Generation and Optimization*, pp. 75–86, IEEE, 2004.

[297] T. Baltrušaitis, C. Ahuja, and L.-P. Morency, *Multimodal machine learning: A survey and taxonomy*, *IEEE Transactions on Pattern Analysis and Machine Intelligence* **41** (2018), no. 2 423–443.

[298] H. Kwon, L. Lai, M. Pellauer, T. Krishna, Y.-H. Chen, and V. Chandra, *Heterogeneous dataflow accelerators for multi-dnn workloads*, in *Proceedings of the International Symposium on High Performance Computer Architecture*, pp. 71–83, IEEE, 2021.

[299] D. Narayanan, K. Santhanam, F. Kazhamiaka, A. Phanishayee, and M. Zaharia, *Heterogeneity-aware cluster scheduling policies for deep learning workloads*, in *Proceedings of the USENIX Conference on Operating Systems Design and Implementation*, pp. 481–498, 2020.

[300] Y. Jiang, Y. Zhu, C. Lan, B. Yi, Y. Cui, and C. Guo, *A unified architecture for accelerating distributed dnn training in heterogeneous gpu/cpu clusters*, in *Proceedings of the USENIX Conference on Operating Systems Design and Implementation*, pp. 463–479, 2020.

[301] Y. Feng and K. Ma, *Chiplet actuary: a quantitative cost model and multi-chiplet architecture exploration*, in *Proceedings of the Design Automation Conference*, pp. 121–126, 2022.

[302] L. Liu, J. Zhu, Z. Li, Y. Lu, Y. Deng, J. Han, S. Yin, and S. Wei, *A survey of coarse-grained reconfigurable architecture and design: Taxonomy, challenges, and applications*, *ACM Computing Surveys* **52** (2019), no. 6 1–39.

[303] D. Scansen, *Chiplets: A short history*, 2021. `https://www.eetimes.com/chiplets-a-short-history/`.

[304] S. Naffziger, K. Lepak, M. Paraschou, and M. Subramony, *2.2 amd chiplet architecture for high-performance server and desktop products*, in *Proceedings of the International Solid-State Circuits Conference*, pp. 44–45, IEEE, 2020.

[305] G. Kenyon, *Heterogeneous integration and the evolution of ic packaging*, 2021. `https://www.eetimes.eu/heterogeneous-integration-and-the-evolution-of-ic-packaging/`.

[306] J. H. Lau, *Heterogeneous Integrations*. Springer, 2019.

[307] A. Tauke-Pedretti, "Common heterogeneous integration and ip reuse strategies (chips)." `https://www.darpa.mil/program/common-heterogeneous-integration-and-ip-reuse-strategies`.