The Trio of Learning, Optimization, and Acceleration for Efficient Electronic Design Automation

HE, Zhuolun

A Thesis Submitted in Partial Fulfillment

of the Requirements for the Degree of

Doctor of Philosophy

in

Computer Science and Engineering

The Chinese University of Hong Kong

August 2023

Thesis Assessment Committee

Professor WONG Martin Ding Fat (Chair)

Professor YU Bei (Thesis Supervisor)

Professor YOUNG Fung Yu (Committee Member)

Professor BAO Yungang (External Examiner)

Abstract

Modern Electronic Design Automation (EDA) is complex and computationally challenging. It consists of a series of difficult optimization problems, accompanied by various analysis and verification processes, which typically take days to weeks to complete. To tackle the efficiency issue in EDA, this thesis proposes several algorithmic methodologies that involve machine learning, customized optimization, and parallel acceleration. Targeted problems span multiple stages from logic synthesis, physical design to physical verification.

Arithmetic block identification in gate-level netlist is an essential procedure for malicious logic detection, functional verification, or macro-block optimization. However, existing methods suffer either scalability or performance issues. To address the problem, this thesis proposes a graph learning-based solution that promises to extract desired logic components from a complete design netlist.

While conventional floorplan approaches rely on metaheuristics, automatic heuristic design through reinforcement learning opens a promising direction for resolving such computationally difficult combinatorial problems. This thesis explores the possibility of acquiring local search heuristics through massive search experience, in contrast to the majority of earlier research that focused on solution construction.

Various neural network processors have been proposed to support the remarkable breakthroughs in deep learning; yet, far fewer discussions have been made on the physical synthesis for such specialized processors, especially in advanced technology nodes. This thesis argues that datapath design is a fundamental methodology in the above procedures due to the organized computational graph of neural networks. As a case study, the thesis investigates a wafer-scale deep learning accelerator

placement problem introduced in ISPD2020 contest in detail.

Design rule checking (DRC) is critical in physical verification to ensure high yield and reliability for VLSI circuit designs. The ever-increasing complexity of contemporary VLSI circuits has necessitated acceleration for computationally intensive DRC tasks in order to achieve reasonable design cycle times. This thesis proposes X-Check, a GPU-accelerated design rule checker, which integrates novel parallel sweepline algorithms that are both efficient in practice and with nontrivial theoretical guarantees. On top of that, the thesis also proposes OpenDRC, which is an efficient open-source DRC engine with hierarchical GPU acceleration.

摘要

現代電子設計自動化 (EDA) 非常複雜且計算上具有挑戰性。 它由一系列困難的優化問題組成，伴隨著各種分析和驗證過程，通常需要幾天到幾週的時間才能完成。 為了解決EDA的效率問題，本文提出了幾種涉及機器學習、定制優化和並行加速的算法方法。 目標問題跨越從邏輯綜合、物理設計到物理驗證的多個階段。

門級網表中的算術塊識別是惡意邏輯檢測、功能驗證或宏塊優化的重要過程。 然而，現有方法面臨可擴展性或性能問題。 為了解決這個問題，本文提出了一種基於圖學習的解決方案，有望從完整的設計網表中提取所需的邏輯組件。

雖然傳統的佈局規劃方法依賴於元啓發式方法，但通過强化學習的自動啓發式設計為解決此類計算困難的組合問題開闢了一個有希望的方向。 與大多數專注於解決方案構建的早期研究相比，本文探討了通過大量搜索經驗獲取本地搜索啓發式的可能性。

各種神經網絡處理器被提出來支持深度學習的顯著突破； 然而，對此類專用處理器的物理綜合的討論卻少得多，特別是在先進技術節點中。 本文認為，由於神經網絡的有組織的計算圖，數據路徑設計是上述過程中的基本方法。 作為案例研究，論文詳細研究了ISPD2020競賽中引入的晶圓級深度學習加速器放置問題。

設計規則檢查 (DRC) 在物理驗證中至關重要，可確保 VLSI 電路設計的高產量和可靠性。 當代 VLSI 電路的複雜性不斷增加，需要加速計算密集型 DRC 任務，以實現合理的設計週期時間。 本論文提出了 X-Check，一種 GPU 加速的設計規則檢查器，它集成了新穎的並行掃描線算法，該算法在實踐中非常高效，並且具有重要的理論保證。 除此之外，論文還提出了OpenDRC，這是一個高效的開源DRC引擎，具有分層GPU加速功能。

Acknowledgments

I would like to express my gratitude to my supervisor, Prof. Bei Yu, for his support and guidance through my Ph.D. study. Prof. Yu is smart, determined, and insightful. He has inspired me to brainstorm research topics, reminded me to keep doing things helpful to my career, and motivated me to become a better person. He has always been giving his students, including me, all-round supports. This thesis would not be possible without his supervision these years.

I would like to thank my thesis committee members for their valuable and constructive comments. Prof. Martin D.F. Wong has let me know how to develop and evaluate research ideas. Prof. Evangeline F.Y. Young has given me numerous advice on different aspects of doing research. Prof. Yungang Bao has provided detailed suggestions to improve this thesis.

I am glad to meet lots of talented and enthusiastic researchers at the department of Computer Science and Engineering, Chinese University of Hong Kong. In particular, Prof. Yufei Tao has brought me to the beauty of computer science. He, as well as Prof. Andrej Bogdanov, Prof. Anthony M.C. So, and Prof. Ken W.K. Ma, is one of the best instructors I have ever seen. Appreciation is for Prof. Yuzhe Ma and Mr. Wei Li as well for their everlasting professionalism and dedication towards research. I need to thank my labmates for many useful discussions between us, thank you Dr. Haoyu Yang, Prof. Tinghuan Chen, Prof. Hao Geng, Prof. Qi Sun, Dr. Ran Chen, Wanli Chen, Chen Bai, Peiyu Liao, Wenqian Zhao, Yuxuan Zhao, Siting Liu, Ziyi Wang, Guojin Chen, Su Zheng, Jiaxi Jiang, Zehua Pei, and Yuan Pu. Besides, I would like to thank my extraordinary fellows who motivate me for lifelong, thank you Dr. Gengjie Chen, Dr. Jordan Chak-Wa Pui, Dr. Bentian Jiang, Dr. Tingyuan Liang, Dr. Runbin Shi, Dr. Yixing Li, Dr. Lixin Liu, Dr. Fangzhou Wang, Mr. Qipan Wang, Mr. Zhisheng Zeng, Ms. Irene Ching-Yun Ko, and Prof. Freda Haoyue Shi. I need to thank other collaborators and experts from academia and industry, thank you Prof. Guojie Luo, Prof. Ngai Wong, Prof. Tsung-Wei Huang, Prof. Yibo Lin, Prof. Biwei Xie, Prof. Xingquan Li, Dr. Yu Huang, Dr. Guangliang Zhang, Dr. Weihua Sheng, Mr. Leo Haisheng Zheng, and Dr. Xun Zhao.

I am very grateful to the souls around who are cheerful, gentle, and talented. It is the strength and power I gain from them that keep me moving forward in the struggle of life. Thank you my friends Dr. Alan Kong, Arthur, Jim, Johnson, and Dr. Daxin Tan. I need to thank my fitness trainer Vinson for the professional trainings that shape my body and mind. I would also like to express my sincere gratitude to my GM, Vicky, for her unwavering tenderness and comforting presence, as she effortlessly smooths out the wrinkles of the days and nights.

Finally, thank you my parents and families. I know you are always standing by my side.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Electronic Design Automation (EDA) refers to a collection of essential software tools for designing *very large scale integrated* (VLSI) circuits. It has long been regarded as the crown jewel of the semiconductor industry. EDA tools are usually arranged in a *design flow*, which chip designers use to design, analyze, and verify the entire circuit in a stage-by-stage manner. In a typical design flow, a circuit is first described with system-level specifications and architectural designs, which are then implemented in hardware description languages. Then, *logic synthesis* implements the circuit in terms of primitive logic gates; *physical design* places the cells on a layout and routes the wire between them. To ensure the validity of the circuit design, various types of analysis and verification along the design flow are required, such as timing analysis, logic simulation, functional verification, and design rule checking. Technically speaking, the aforementioned procedures entail solving various computationally challenging problems, including mathematical optimization, boolean function simplification, graph isomorphism, longest/shortest paths, computational geometry, deconvolution, and so on. In particular, many of these problems are known to be NP-complete, which are unlikely to be solved optimally in a reasonable amount of time.

In recent decades, chips have undergone rapid evolution, growing in size, power, and efficiency (Ma, 2020). But as technology nodes advance, EDA tools face scalability issues as they deal with larger-scaled problems with more intricate constraints and objectives. At the same time, improving tool/algorithm efficiency has been a key concern in the advancement of electronic design

automation, which has received extensive attention and abundant efforts. New methodologies, which we will introduce in the following section, are believed to provide promising directions to address the efficiency issue in the modern computing era.

1.1 Challenges and Opportunities

As mentioned above, efficiency is a critical concern for electronic design automation. The challenges we face include the following:

- Firstly, there are billions of transistors in today's extremely large circuits. An algorithm running in $\Omega(n^2)$ time will be unacceptably slow given such input scale. Considering the hard nature of EDA problems, we inevitably need new methodologies to address them.
- Secondly, existing algorithms often have limited scalability and parallelizability. Although classic methods are proven effective in the circumstances at that time, how to orchestrate them with modern, highly parallel compute substrates remains an open problem.
- Thirdly, typical EDA methods are for general circuit designs. Special considerations have to be taken when adapting the current design flow to specific designs and targets.

Fortunately, the development of artificial intelligence and the relevant computing facilities have given us new weapons to combat the above issues. Specifically, machine learning demonstrates brand-new paradigms to cope with difficult and even hard-to-formalize problems by mimicking solutions with black-box neural models. General-purpose graphics processing units (GPGPUs), the primary booster behind the blossom of AI, offer enormous computational power for highly parallel applications. On the other hand, as AI applications become more prevalent, customizing EDA methodologies for neural network processors turns out to be an effective strategy to push circuit performance to the limit.

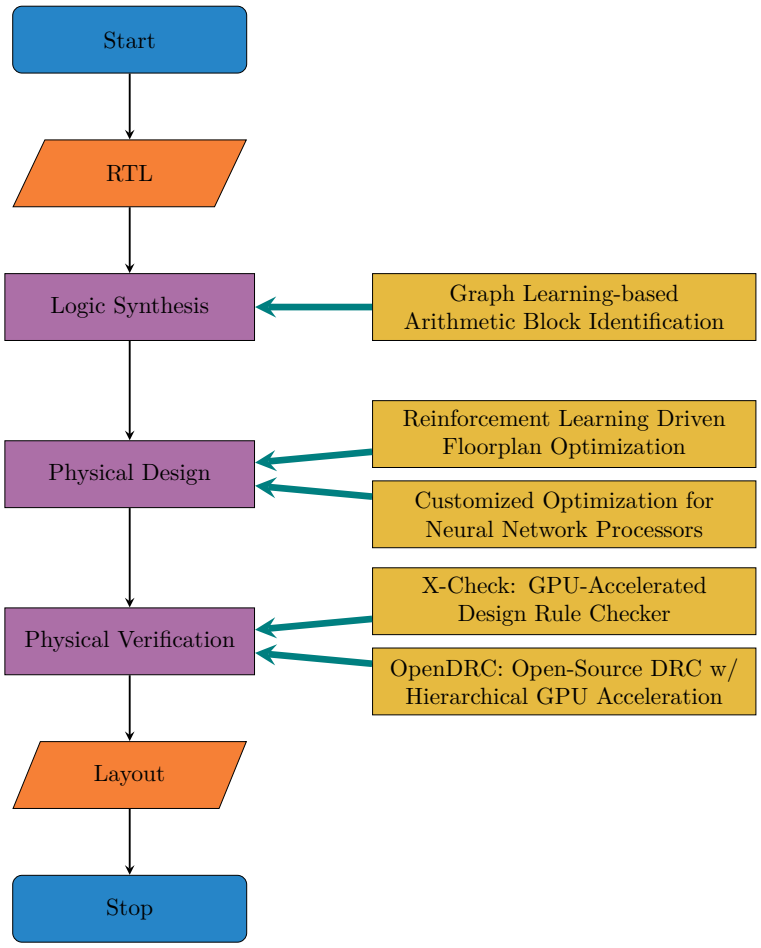


Figure 1.1: A typical design flow and the proposed methodologies in the corresponding design stages.

1.2 Thesis Overview

This thesis aims to tackle the efficiency issue in electronic design automation by proposing several essential algorithmic methodologies in the new computing era, involving machine learning, parallel computing, and customized optimization. Targeted problems span multiple design stages from logic synthesis, physical design to physical verification. Figure 1.1 illustrates a typical design flow and the proposed methodologies in the corresponding design stages.

In Chapter 2, we review existing solutions and approaches for relevant design stages, including arithmetic block identification, floorplanning, placement, and design rule checking. In particular, we focus on promising methodologies that help to improve efficiency, such as datapath driven placement,

reinforcement learning driven methods, and parallel computing. These methodologies form the basis of the presented research outcome in this thesis.

In Chapter 3, we propose a graph learning-based framework that promises to extract desired logic components from a complete design netlist. We also customize graph neural network (GNN) architecture dedicated for netlist representation learning, which outperforms standard GNN variants in our tasks.

In Chapter 4, we explore the possibility of acquiring local search heuristics through massive search experience. To demonstrate the applicability, we target the floorplan problem, and present methodologies to train an agent that is able to automatically optimize floorplan solutions through local search.

In Chapter 5, we investigate a wafer-scale deep learning accelerator placement problem in detail. We especially argue that datapath design is an essential methodology in the above procedures due to the organized computational graph of neural networks.

In Chapter 6, we present novel parallel algorithms and implementation details for efficient design rule checking (DRC). We show that lots of DRC tasks can be solved with a general prefix computation scheme, which can be parallelized with both nontrival theoretical guarantee and high efficiency in practice. On top of the algorithms, we propose two GPU-accelerated design rule checkers, X-Check and OpenDRC. As a side product, we further discuss the development of an STL-like programming library for GPUs.

We summarize and conclude the thesis in Chapter 7.

Chapter 2

Literature Review

2.1 Arithmetic Block Identification

Arithmetic block identification is to locate arithmetic macros (e.g., adders, multipliers) in a gate-level netlist. Classic approaches to identify arithmetic components can be roughly categorized into either *structural methods* (Doom et al., 1998; Rubanov, 2006; Li et al., 2013; Subramanyan et al., 2013a; Meade et al., 2016) or *functional methods* (Subramanyan et al., 2013b; Gascón et al., 2014).

Structural methods concentrate on circuit topology while paying little attention to the functionality of the circuit (Azriel et al., 2019). For instance, *shape hashing* is introduced by Li et al. (2013) to group wires with the same local topology together to form candidate words. Specifically, a k-step-bounded depth-first traversal of the graph is performed starting from each wire to produce its serialization using the gate and wire types. Some other works consider the scenario where a library of reference circuits is given, and the problem becomes mapping subcircuits with reference circuits. Rubanov (2006) formulat the subcircuit matching problem as a regularized quadratic assignment problem to minimize both graph distance and vertex label distance. A nonlinear version of the iterative Kaczmarz Method (KM) is used to solve the obtained equations. Structural methods usually promise to identify target blocks with customized algorithms efficiently. However, they are often mathematically incomplete due to the heuristic methodology.

On the other hand, **functional methods** functionally analyze the circuit for potential arithmetic

components. A typical example as by Subramanyan et al. (2013b) extends the above shape hashing method by considering the functions implemented by a set of gates using cut enumeration. They enumerate all 6-feasible cuts and group equivalent cuts using permutation-independent Boolean matching. In this way, each equivalence class of cuts may match a known library function. It is further proposed by Subramanyan et al. (2013b) to use functional verification tools for module matching: suppose C is a potential arithmetic block with input word X and side inputs Y, C' is a reference circuit, and Φ is an inserted comparator miter between the outputs of C and C', the Quantified Boolean Formula (QBF) $\exists Y \forall X \Phi(X, Y)$ exactly models the equivalence checking problem. As can be seen, functional methods are accurate and solver-ready at the cost of ultra-long runtime.

The development of machine learning and deep neural networks has provided alternate solutions to recognition and classification. To efficiently identify functional units, Silva et al. (2018) and Fayyazi et al. (2019) recently propose deep learning-based solution to recognize arithmetic circuits. Silva et al. (2018) develop a flow that converts conjunctive normal form (CNF) clauses into images, which are later rescaled to the desired size and fed into deep learning classifiers. Fayyazi et al. (2019) present a compact representation called existence vector (EV) that encodes a circuit node with its all neighbors. A fixed number of EVs are selected to satisfy the fixed-input-size requirement of convolutional neural networks. However, these solutions are dedicated to one given unknown functional block. When dealing with large-scale netlist design, these solutions are facing significant challenges.

2.2 Floorplan

Floorplanning is the first step in physical synthesis, which aims to roughly determine geometric relationship among circuit modules and to estimate the cost of the design. Various data structures are introduced for the representation of the geometric relation. A slicing floorplan, where the whole design can be recursively divided horizontally or vertically until each part contains only one module, is naturally encoded by a binary tree, whose internal nodes are for the horizontal or vertical cuts and the leaves denote the modules. Equivalently, polish expression is used (Wong and Liu, 1986) to encode the postfix of the same binary tree. As for general floorplans without a slicing structure (i.

non-slicing), many other elegant representations are invented, e.g., O-tree (Guo et al., 1999), B*-tree (Chang et al., 2000), Sequence Pair (Murata et al., 2003), and Twin Binary Sequences (Young et al., 2003). With a flexible and effective representation, good floorplan results could be achieved through constructing or perturbing a data structure in a systematic way (Guo et al., 1999), by heuristics (Cong et al., 2005), or by some means of meta-heuristics like *genetic algorithms* (Lin et al., 2002) or *simulated annealing* (Chen and Chang, 2005).

2.3 Datapath Driven Placement

We present in this section some useful techniques in datapath driven placement.

2.3.1 Placement with Datapath Constraints

The idea of datapath driven placement can date back to no later than the work by Cai et al. (1991) published in the year of 1990, which considers automatic generation of bit-sliced datapaths in high performance DSP circuits. The datapath consists of multi-bit operators called functional building blocks (FBBs), such as adders or registers. The proposed linear placement tool generates a linear ordering of the FBBs to minimize the layout area. In their work, the ordering solution space is represented as an acyclic directed graph, so that the orderings can be searched with the A* algorithm. The algorithm achieves good performance and runs much faster than metaheuristics (e.g. simulated annealing), and the authors emphasized that the algorithm is flexible in adapting various cost functions.

Later on, datapath driven standard cell placement was proposed (Tsay and Lin, 1995). In this work, strongly connected subcircuits (i.e., cones) are extracted by a breadth-first algorithm augmented with heuristic rules. These cones are treated as soft macro cells, and are placed by a macro-cell placement algorithm to reduce the intercone wiring length. Macros are converted back to cells by a mapping subsystem to preserve the topological relationship between them.

It is argued that if the datapath is generated separately and simply merged with netlists of other parts, the placement tool has little control of the exact location where a cell might be placed (Ye and De Micheli, 2000). In this way the regularity information will be lost. Given that, the

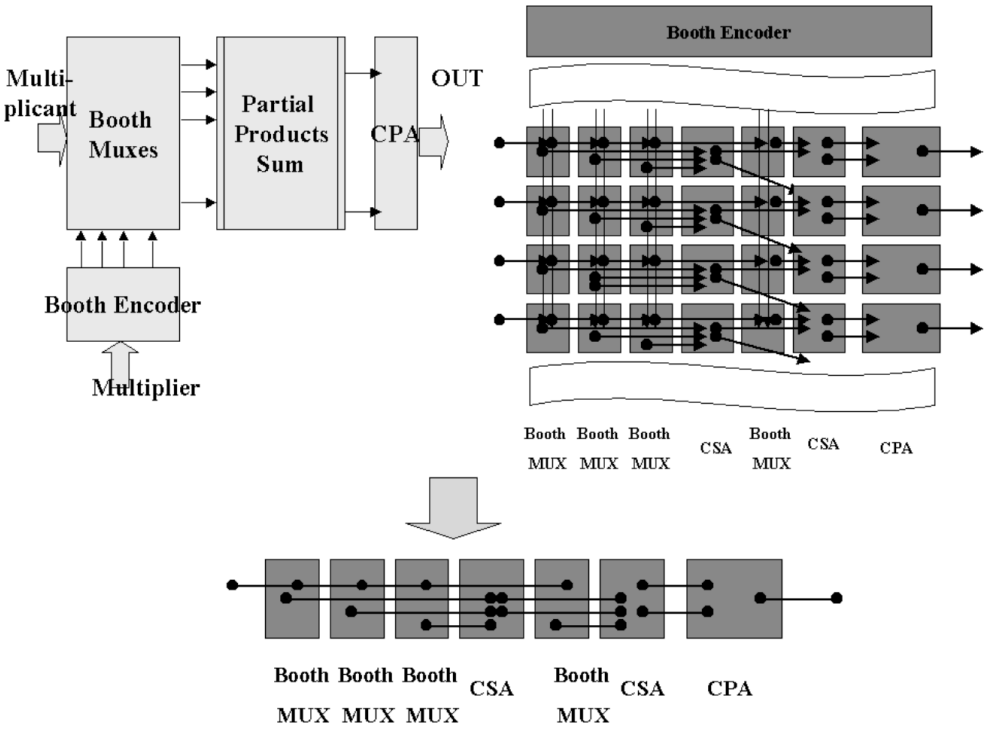


Figure 2.1: Abstract physical model is a bit-sliced abstraction of a datapath circuit. The figure illustrates the APM of a booth multiplier (Ye and De Micheli, 2000).

authors proposed an abstract physical model (APM), a bit-sliced abstraction of a datapath circuit. Figure 2.1 demonstrates the APM of a booth multiplier (adopted from Ye and De Micheli (2000)). The APM is compiled from HDL, and the blocks in APM are placed abutted to each other. Since the linear placement problem is NP-hard, they proposed a two step heuristic that first determines an initial ordering by a quadratic optimization procedure, and then a sliding window optimization is performed to solve wirelength and congestion violations.

Datapath has been considered in detailed placement (Serdar and Sechen, 2001), where a modified O-tree (Guo et al., 1999) based placer is able to place components on reflection lines while obeying design rules. It is also considered in physical design inside SOC (Tong et al., 2002; Jing et al., 2002), and for parallel multiplier design (Bae et al., 2015). A lot more works for datapath driven general ASIC design have been presented. Ye et al. (2002) place datapath clusters with constraints that 1) the relative locations of the clusters should follow the dataflow order and 2) the relative orientations of the clusters should follow the bit order, and the same bit order should be preserved throughout

the dataflow. The authors name it '1.5 dimensional placement' and proposed to solve it by linear placement heuristics similar to Ye and De Micheli (2000). A sigmoid-function-based density model is proposed (Chou et al., 2012) for separate optimization in horizontal and vertical directions. Blocks of each functional stage align vertically due to the regularity in datapaths, which reduces variables in the optimization problem. Wang and Shin (2017) place datapath macros with other random blocks by an analytical placement algorithm, while the relative location of bit-slices can be adjusted inside the datapath macros to optimize total wirelength. Results have shown that these techniques produce better results in wirelength (HPWL, StWL) and/or routability.

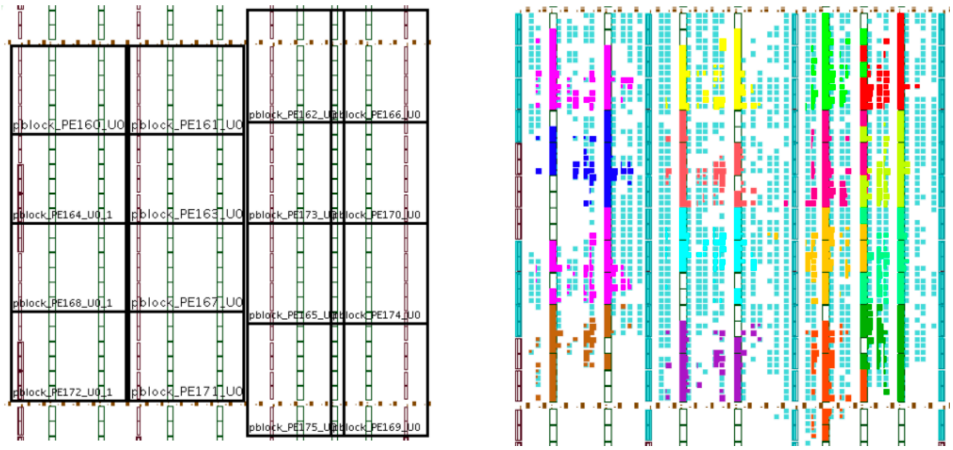
Systolic arrays are a popular choice to support neural network computations due to their regular topology and simple interconnections. Despite the layout-friendly structures, it was reported that current FPGA CAD tools are unable to synthesize systolic arrays in high quality (Zhang et al., 2019c). One of the key reasons might be that DSPs are distributed into columns over the whole FPGA chip. Meanwhile, there are around 15× more intra-PE nets than inter-PE nets, optimizing the length of which result in a distorted layout. Given the above, the authors propose to perform floorplanning and set placement constraints to restrict fixed locations for PEs. To enable the topology-aware floorplanning, the region in which the PEs are placed should provide sufficient hardware resources, and should be as close as possible to the used I/O banks. Then the PEs are mapped to the available DSP columns by enumeration. Figure 2.2 shows the placement result of PEs with floorplan constraints (adopted from Zhang et al. (2019c)). The authors reported 1.29× frequency improvement and 1.5TOP/S in deploying a VGG model onto the Xilinx KCU1500 platform.

Benchmarks are released (Ono and Madden, 2005; Ward et al., 2011) to evaluate the performance of the placers.

2.3.2 Methods for Regularity Extraction

Along with the methodologies in datapath driven placement, various approaches for regularity extraction have been proposed. We review some of the representative approaches in this section.

Intuitively, as described by Nijssen and Jess (1996), consider cells associated with the same bit-*slice* are lined up horizontally, and the cells of the same type occurring at similar places are stacked alongside forming *stages*. The circuit is thus fitted onto a matrix of rectangular buckets that yields



(a) (b)

Figure 2.2: PE placement with floorplan constraints. Floorplanning re-organizes PEs more regularly (Zhang et al., 2019c).

maximum density cell placement. In addition to the above *geometric regularity*, the *interconnect regularity* indicates that almost all nets are contained either within one slice or within one stage. Nijssen and Jess (1996) define a local regularity metric by interpreting the distribution of the number of pins, and a regularity extraction algorithm is proposed by expanding search-waves through the network, stage by stage according to the regularity metric.

A signature-based regularity extraction algorithm is proposed later (Arikati and Varadarajan, 1997). The *signature* of a random instance is dictated by its master cell and its connectivity to datapath instances. Then a connectivity cost function is defined based on some objective, such as the vertical distance between two pins. The random instances are sorted based on the signatures and are partitioned into blocks with the same signature. Finally, the regular functions are generated taking the connectivity cost into consideration. The authors also propose a relaxed function-based signature.

Covering a circuit by templates is another line of research. Besides assuming a library of provided templates, Chowdhary et al. (1999) present an approach to automatically generate all possible templates for the input circuit. Despite the inherent difficulty in template generation (which is similar to enumerating isomorphic subgraphs), they propose to extract only maximum degree of regularity (assumption 1 in their paper), and to assign incoming edges a unique index (assumption 2

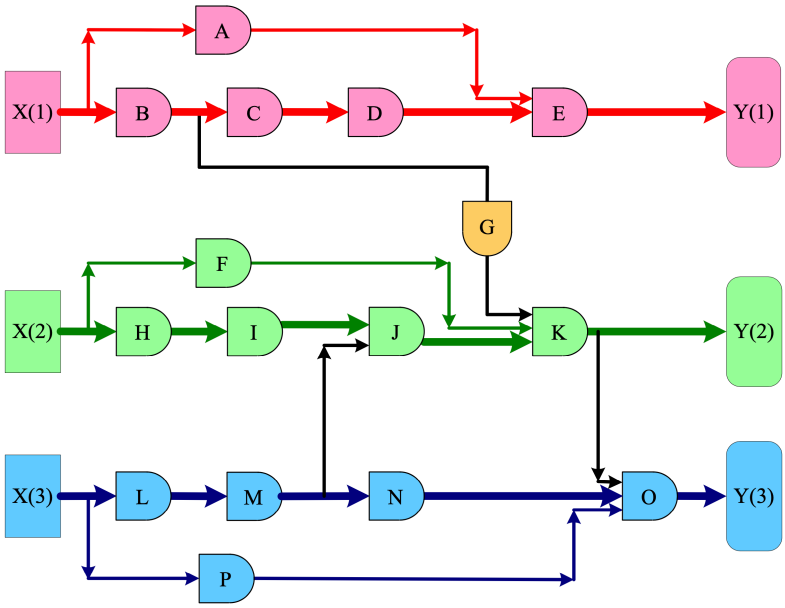


Figure 2.3: The problem of datapath main frame identification can be transformed to a network flow problem (Xiang et al., 2013).

in their paper). With the two assumptions, the number of possible tree templates is reduced to within V^2 from 2^V. The template generation algorithm is then extended to generate multi-output function. The authors also propose two heuristics to effectively cover the graph by templates, including *largest-fit-first* that selects the template with the maximum area, and *most-frequent-fit-first* that selects the template with the maximum number of subgraphs.

The regularity extraction problem can be converted to a network-flow problem. Xiang et al. (2013) define *datapath main frame* (DMF) as a set of n disjoint paths from input to output such that the number of datapath gates on these paths is maximized. To identify DMF, an optimal network-flow based algorithm is presented. Basically, capacities U and costs C are assigned to the network graph, where capacity is 1 for all the nodes and edges, and the cost for nodes is a constant negative number, and 0 for the edges. The min-cost max-flow algorithm will be applied to the flow network, which guarantees the optimality with polynomial runtime. Figure 2.3 demonstrates the data main frame identification problem (adopted from Xiang et al. (2013)).

Properties of bit-slices include 1) small area variance (similar), 2) large area mean (long), and 3) minimized overlaps (Huang et al., 2017). In the same paper (Huang et al., 2017), the authors propose

a two-stage method that first optimize 1) and 2) with a balanced bipartite edge-cover algorithm, and minimize 3) with simulated annealing. Since every bit-slice path is a bit line connecting I/O vectors, a datapath is modeled as a bipartite graph, where the vertices correspond to either larger I/O vector or narrow I/O vector, and the edges correspond to the bit lines. Therefore, the problem of extracting feasible set of bit-slice paths is transformed to the problem of covering vertices in the bipartite graph.

It is worth mentioning that these techniques have also been utilized in logic synthesis (Kutzschebauch and Stok, 2000; Kutzschebauch, 2000; Rosiello et al., 2007). However, it was also pointed out (Ienne and Grießing, 1998) that extraction of regularity from synthesized netlists is difficult and requires counterproductive simplifications to the synthesis process.

2.3.3 Machine Learning Techniques

Recently, machine learning techniques are utilized for datapath extraction (Ward et al., 2012a). Specifically, graph-based (e.g., automorphism) and physical features (e.g., cell area) are analyzed and extracted from the netlist. These features are fed to support vector machine (SVM) and neural networks (NNs) to classify and evaluate datapath patterns. Both SVM and NNs are to maximize the evaluation accuracies of datapath and non-datapath patterns, which are defined as the rate of correctly detected datapath/non-datapath patterns over the total number of corresponding structures. The proposed placer, PADE, has demonstrated reasonable improvement in wirelength. Figure 2.4 shows placement result by PADE that effectively handles the datapath (adopted from Ward et al. (2012a)). The authors also proposed a unified placement flow (Ward et al., 2012b, 2013) that handles both random logic and datapath standard cells on top of a force-directed placer. Several techniques for structural-aware placement, such as skewed weighting for net alignment and bit-stack aligned cell swapping, are proposed and discussed in the paper.

2.4 Efficient Design Rule Checking

The continuing and growing high computational costs of DRC drive us to pursue new computing techniques to reduce the turn-around time for these tasks. From the methodology perspective,

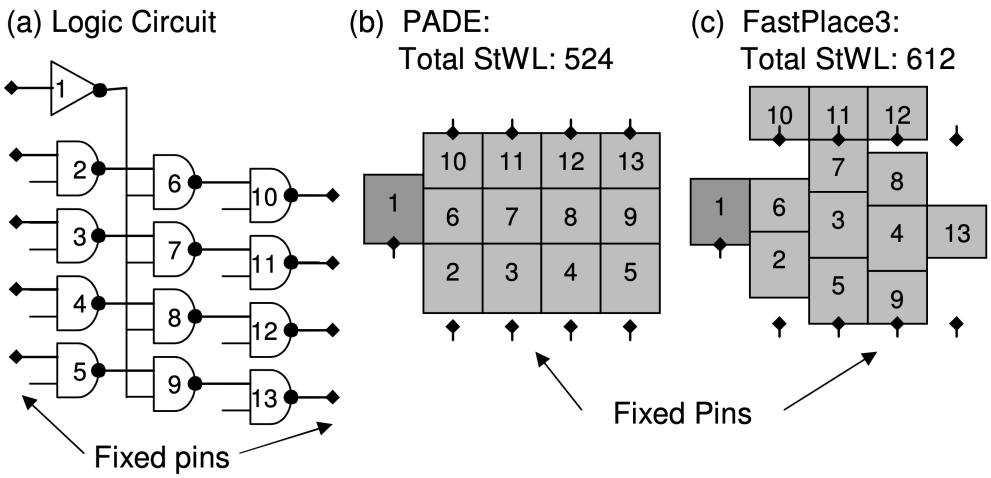


Figure 2.4: PADE effectively handles datapath in placement. Adopted from Ward et al. (2012a).

these attempts could be classified into three categories, namely 1) to design better algorithms, 2) to parallelize computation workloads, and 3) to approximate desired results.

As for DRC algorithms, many theoretical results are obtained a few decades ago, such as rectangle intersection report (Bentley and Wood, 1980), line segment intersection report (Chazelle and Edelsbrunner, 1992), orthogonal range query (Willard, 1985), and boolean mask operations (Lauther, 1981). Proper data structures to cope with layout data are also discussed, including binary space partitioning data structures like quad-tree (Finkel and Bentley, 1974) and kd-tree (Bentley, 1975), hierarchies of bounding volumes like r-tree (Guttman, 1984) and its variants, and other customized data structures like corner stitching (Ousterhout, 1984). These historic milestones form the algorithmic foundations of today's design rule checkers.

Parallel computing carries out computation workloads in multiple processors simultaneously to reduce turnaround time. One standard technique is to partition the layout into tiles and perform DRC on the tiles in parallel, which has been investigated since the 1980s (Bier and Pleszkun, 1985). Similarly, region-based methods (Nandy, 1994; Hsu et al., 2011) partition the circuit into subregions for spatial parallelism; When the layout is equipped with hierarchical information, it is also possible to exploit cell-level parallelism from the hierarchical representation, as illustrated by Gregoretti and Segall (1984); Hsu et al. (2011). At the edge level, Carlson and Rutenbar (1988, 1991) have proposed parallel algorithms for Manhattan geometry (Carlson and Rutenbar, 1988) and general

(oblique) geometry (Carlson and Rutenbar, 1991). The above approaches can be considered as *data parallelism*. For *task parallelism*, since design rule checking often involves more than one algorithm, Marantz (Marantz, 1986) has developed a parallel checker that runs different checking algorithms concurrently. A systematic approach that combines both data and task parallelism is presented in (MacPherson, 1995). These works gain benefits from different hardware platforms, including SIMD engines (Koranne, 2004), specialized hardware (Kane and Sahni, 1984; Luo et al., 2000), and distributed systems (Nandy, 1994; Pais et al., 2001).

Approximation methods sacrifice result accuracy to trade for improved runtime efficiency, among which machine learning (ML) algorithms are a popular subset. By predicting design rule violation (DRV) types of clipped layout regions, ML-based design rule checkers have demonstrated tens of times speedup compared with an accurate checker (Francisco et al., 2020). In the design stages (e.g., in placement or routing), ML is widely used to predict DRC hotspots (Zeng et al., 2020) and the number of DRVs (Xie et al., 2018), without locating and identifying exact violations (Francisco et al., 2020). Although not directly accelerating the checking process, some ML-enhanced DRC schemes are worth mentioning, such as design rule verification (Alam et al., 2023), design rule augmentation (Dai et al., 2009), and DRC script generation (Zhu et al., 2022).

2.5 Reinforcement Learning for Combinatorial Optimization, Local Search Algorithms, and Physical Design

Reinforcement learning interacts with the environment and trains an agent to survive, which can in principle create new knowledge about the space which the agent live in. In addition to the breakthroughs in game playing (Silver et al., 2017) and robotic control (Gu et al., 2017), the community also spent efforts in the discipline of combinatorial optimization, local search algorithms, and physical design.

2.5.1 RL for Combinatorial Optimization

An end-to-end *actor-critic* training framework (Bello et al., 2016) based on the *pointer network* architecture (a variant of recurrent neural network) is proposed to tackle the Travelling Salesman

Problem (TSP). Later on, *structure2vec* (a graph embedding network) and the off-policy *Q-learning* are utilized (Khalil et al., 2017) to solve TSP and other optimization problems over graph. Apart from TSP, researchers have also achieved significant results on Vehicle Routing Problem (VRP) (Kool et al., 2019), Job Scheduling (Chen and Tian, 2019), and Satisfiability Problem (SAT) (Yolcu and Póczos, 2019).

2.5.2 RL for Local Search Algorithms

The idea of enhancing local search heuristics with reinforcement learning is not that new. Researchers seek to boost the local search by selecting a good starting point (Boyan and Moore, 1998; Zhou et al., 2016), by tuning search parameters (Benlic et al., 2017), by scaling a regularization term (Beloborodov et al., 2020), or by switching between heuristics on the fly (Nareyek, 2003). Basically, all these work adopt an existing local search algorithm, and improve a few settings of that algorithm with reinforcement learning.

2.5.3 RL for Physical Design

Placement

Macro Placement Reinforcement Learning has been used in chip macro placement (Mirhoseini et al., 2020). In this work, the target is to place a netlist graph of macros (e.g. SRAMs) to a chip canvas to optimize PPA. Overall, the method is to first sort the macros in descending size, which are placed sequentially by the agent, and followed by force-directed standard cell clusters placement and greedy legalization. Therefore, in their reinforcement learning settings, each state is a possible partial placement, and each action is to place a macro onto the canvas. The reward is always 0 except the last action (that leads to the terminal state), where the reward is the negative weighted sum of proxy wirelength and congestion of the placement.

Distilling useful features from state representation is essential for good decision making. In the policy network, the netlist is encoded with a graph convolutional network (GCN) for macro embedding and edge embedding, which are concatenated with other metadata. A key intuition presented in the paper is that a policy network capable of transferring placement knowledge across

chips should first be able to encode the features into a meaningful signal in inference time. Given that, they proposed to first train the model for reward prediction in a supervised manner, with a dataset of 10000 chip placements of 5 netlists. After the supervised training, the reward prediction layers are removed, and the encoder component is used in the policy network.

To handle different grid size, they allow a maximum row/column number of 128, and the L-shaped unused section are masked if the grid size is smaller. The decoder is composed of deconvolution layers and batch normalization layers to generate a probability distribution for the actions. The RL agent is trained with PPO (Schulman et al., 2017) with a clipped objective:

$$L^{\text{CLIP}}(\theta) = \hat{\mathbb{E}}_t[\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)], \tag{2.1}$$

where $\hat{\mathbb{E}}_t$ is the expected value at time step t, r_t is the probabilistic ratio between new policy and old policy, and $\hat{A}_t$ is the estimated advantage at time step t.

During the evaluation, the policy network without further tuning yields a placement result (which they called zero-shot placement) in less than one second. They observed that a pre-trained model with 2 hours fine-tuning performs even better than another model trained 24 hours from scratch, which indicates the effectiveness of offline training. In comparison with baseline methods, the RL method converges in 3 to 6 hours, which is much faster than simulated annealing (18 hours) and slower than RePlAce (Cheng et al., 2019) (1 to 3.5 hours), one of the state-of-the-art placer. They also compared with human experts (a chip design team) that involves many iterations over a few weeks. When it comes to the placement quality, the RL method is able to consistently meet timing and congestion constrains, and outperforms human experts in several metrics including WNS, area, power, and wirelength.

Placement Heuristic Selection Another work worth mentioning on RL-enhanced placement is (Murray and Betz, 2019). In this work, the RL agent is trained to select the move type in an SA framework for Field-Programmable Gate Arrays (FPGA) placement. This is naturally modeled as a multi-arm bandit problem with a Q table for each action (possible move type). Given that, the agent selects the move type with the highest Q-value and samples a random move, which the SA engine decides to accept or reject. The proposed technique was integrated into VTR 8 placer (Betz

and Rose, 1997) and achieved improved quality-runtime tradeoff.

Routing

Global Routing Global routing partitions the routing region into tiles and decides tile-to-tile paths for all nets while attempting to optimize some given objective function (e.g., total wirelength and circuit timing) (Wang et al., 2009). In (Liao et al., 2020b), RL is used for sequential single net routing. In their settings, the agent observes a vector representing current coordinate, target pin coordinate, and the available resource of the neighbouring tracks, and picks a direction (out of the 6 possible) to move. The reward is $+100$ if the target pin is reached, otherwise -1 is given.

Since the action space is discrete and finite, they selected to use DQN (Mnih et al., 2013) for training. Before the exploration by RL agent, they collect samples by running A* algorithm to fill the experience replay buffer (which they called memory burn-in). The evaluation was conducted on toy examples like $8 \times 8 \times 2$ grid graphs with only a few nets. One interesting observation is that A* performs much better in easy cases (i.e., no edge with positive capacity is fully utilized in A*), while RL agent wins more in difficult cases.

Detailed Routing Given global routing paths, detailed routing determines the exact tracks and vias for nets (Wang et al., 2009). In a track-assignment detailed router, a track assignment step is performed before detailed routing to place the long routes onto tracks defined by the width space pattering (WSP), which reduces the search space for the router as it only needs to connect the components (i.e., instance terminal) of the same net together. In (Liao et al., 2020a), RL is used to decide the track assignment order.

Specifically, the attention-based encoder-decoder framework (Kool et al., 2019) is adopted, where greedy rollout is used as a baseline to train the agent with REINFORCE (Williams, 1992). The observation of the agent includes an overlap graph of instance terminals, in which an edge indicates two terminals overlap in x-range and hence cannot be assigned to the same track, and an assignment graph, which is a bipartite graph of instance terminals and available tracks.

The evaluation was done on two artificial datasets of analog design problems. The small dataset contains placement solutions for Comparators and OpAmp, which consists of 10-100 instance ter-

minals, while the large dataset are designs for analog-to-digital converter (ADC) with 100–1000 instance terminals. In comparison with the genetic algorithm (GA), the RL solution runs 100× to 1000× faster, while GA outperforms in solution quality. The paper also shows that the performance of RL agent could be further improved with more training samples.

Printed Circuit Boards Routing To target printed circuit boards (PCBs) routing, (He and Bao, 2020) proposed to use Monte-Carlo-tree-search (MCTS) to guide the agent training. Basically, a search tree is constructed based on rollouts to obtain optimal solution, which is then back-propogated to update the agent parameters. The input of the agent is the grid graph (a matrix) states, and the output is one of the four actions representing the four directions to go.

In practice, they consider a single layer board of 30×30, so the model architecture is a convolutional neural network (CNN). To train this model, a dataset of 2000 randomly generated circuits (grid size 30-by-30) routed with maze routing and manual routing was created. Then each vertex on the net of the circuit can be taken as the head of a training sample. With one sample taken from each net, totally 9459 samples were obtained. Experiments then show that the agent is able to obtain good solutions after thousands of training iterations, while some instances could be successfully routed by A* or Lee's algorithm (because of the routing order).

The authors have also emphasized to use maximal reward in MCTS expansion (Max-UCT), instead of using average reward (Avg-UCT) as in vanilla MCTS. Experimental results show that using Max-UCT indeed greatly outperforms Avg-UCT in terms of wire redundancy ratio and convergence speed. One possible explanation, as pointed out by the authors, is that the routing search space is very large and nice solutions are rarely seen, which makes average reward somehow useless in the context.

Sizing

Transistor sizing can be modeled as a continuous parameter search problem. In (Wang et al., 2018), a sequence to sequence model is used to encode observations (e.g. voltage at a node) and to decide sizing solution (e.g. width and length of transistors, capacitance of capacitors). Reward function was designed to include both hard constraints and optimization targets. To train the model,

DDPG (Silver et al., 2014) was used, with truncated uniform distribution as noise for search space exploration. Experiments on two transimpedance amplifiers circuits show that the proposed method can achieve comparable performance with much higher (250×) sample efficiency compared with grid search based human design. It also outperforms Bayesian Optimization under the same runtime constraint.

Later on, GCN was involved (Wang et al., 2020) since the circuit topology can be naturally represented by a graph, whose vertices are transistors and edges are wires. In this work, the agent determines the action for each node (e.g., width, length, and multiplexer for an NMOS), and therefore the action decoder is component-specific. The state for each node includes one-hot representation for transistor index and component type, as well as a vector of selected features. Note that the action space is continuous here, otherwise the action space will be too large to deal with. The reward is still figure of merits (FoM), a weighted sum of normalized performance metrics. The effectiveness of RL agent is demonstrated on a few real-world circuits, compared with black-box optimization tools and human experts. Experiments on knowledge transfer between nodes and between topologies are also conducted, which shows that transfer learning indeed helps, and using GCN for feature extraction is critical.

2.6 Methodologies for GPU-enabled EDA

Recent years have seen GPU acceleration for various design automation stages to speed up design closure. It is pointed out that many conventional parallel algorithms do not scale beyond a few CPU cores (Guo et al., 2021a), and how to better utilize the massive computing resources in GPUs needs special considerations. We detail two popular methodologies to develop efficient GPU-enabled applications.

The first one is **to cast a design automation problem into another problem solvable by current tools/infrastructure**. One of the most clever ideas is DreamPlace (Lin et al., 2020), where the analytical placement problem is converted to neural network training and hence can be implemented on top of the PyTorch framework. Similarly, GATSPI (Zhang et al., 2022) is a GPU-enabled gate-level simulator developed with DGL/PyTorch and customized CUDA kernels; in

particular, netlists are transformed into graph objects for further operations. FastGR (Liu et al., 2022) regards the batched net routing ordering problem as a task scheduling problem, which is solved using the Taskflow (Huang et al., 2021) scheduler. By utilizing existing solvers or frameworks, developers could focus more on problem formulation and algorithm customization, without needing to build everything from scratch.

The second methodology is **to design novel GPU-friendly computation kernels for some critical tasks in the design flow**. Guo et al. (2021b) decompose density accumulation, an essential primitive in placement, into a density allocation phase, plus a 2D *prefix sum* phase, which is easily parallelized. GAMER (Lin et al., 2021) solves the shortest path problem in routing by iterative vertical and horizontal sweeping/relaxation, which is also conceptually a *scan* process. Guo et al. (2020) analyze and implement GPU-friendly algorithms for timing analysis, including a breadth-first search for RC delay computation, parallel levelization by advancing 'frontier', and table lookup/interpolation.

We refer readers to Lin (2020) for a survey on GPU acceleration in VLSI back-end design. Moreover, GPU acceleration is also a popular topic in conference contests (Zhang et al., 2020; Pasandi et al., 2021).

2.7 Advanced Technologies for Neural Network Processors

Advanced technologies have shown great potential to address scaling challenges. Due to the page limit, we mention a few of them and refer readers to (Ielmini and Ambrogio, 2019) for a more comprehensive survey.

Processing-in-memory (PIM) provides massive parallelism with high energy efficiency (Wang et al., 2019b), offering new solutions to address challenges in modern computer systems. Recent work have demonstrated that neural network computation can be implemented in various emerging non-volatile memories (NVM), such as RRAM (Chi et al., 2016; Sun et al., 2018), STT-MRAM (Yan et al., 2018; Pan et al., 2018), PCM (Kim et al., 2020), and memristor (Yao et al., 2020). In-memory analog simulation is another promising approach, for instance (Li et al., 2018) based on memristor crossbar and (Li et al., 2020) based on FTJ. Without the need for moving data between memory and

processor, these accelerators substantially improve the performance and efficiency of neural network execution. However, lots of systems rely on an external control device that may reduce the benefit of PIM.

The nanophotonic circuit is an alternative neuromorphic computing system due to its ultra-high bandwidth, speed and ultra-low energy consumption. In nanophotonic, signals are encoded in the amplitude of optical pulses propagating in the photonic integrated circuit, which implements a multi-layer optical neural network (ONNs) (Shen et al., 2017) . Recent advances include exploring nonlinear functions with ONNs (Zuo et al., 2019) and reducing area overhead of ONNs (Gu et al., 2020).

Different 3D technologies are available to offer a wide spectrum of integration schemes. A neural network accelerator Tetris (Gao et al., 2017) implemented with through-silicon-via (TSV) 3D memory achieves 4.1× and 1.5× performance and energy improvements. Specifically, the memory substrate of Tetris is hybrid memory cube (HMC) (Jeddeloh and Keeth, 2012), one of the well-known realizations of 3D memory (another is high bandwidth memory (HBM) (Lee et al., 2014)). ThruChip Interface (TCI), an alternative to TSV, is a high-performance wireless vertical interconnect technology used to transmit signals between multiple stacked dies. QUEST (Ueyoshi et al., 2018) is a DNN inference engine stacked with multiple SRAMs using TCI, which enables large memory capacitance. Monolithic 3D (M3D) IC technology has shown its potential to address the power, performance and area (PPA) scaling challenges (Chang et al., 2017). The schematic in Figure 2.5 shows a gate-level monolithic 3D (adopted from Chang et al. (2018)). It is also studied whether M3D can benefit deep learning hardware design (Chang et al., 2017, 2018), where a Gaussian Mixture Model for acoustic modeling (Su et al., 2010) is mapped to both 2D and M3D designs for comparison. To implement two-tier full-chip M3D designs, the design flow proposed by Panth et al. (2014) is adopted. The flow starts with scaling width and height of all standard cells and metal layers by $1/\sqrt{2}$, so that an overlap-free design can be implemented in half footprint of the corresponding 2D design. This shrunk design is synthesized and the cell placement information (i.e., x-y location) is obtained. Then the design is scaled back to the original size, and the overlapped cells are partitioned into two tiers. The partitioning is based on an area-balanced min-cut algorithm, and the remaining overlapping (on the same tier) are removed through legalization. The connections between both tiers are called mono-

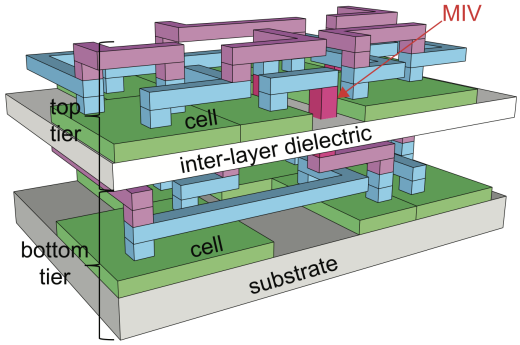


Figure 2.5: A schematic of monolithic 3D. Transistors are fabricated onto multiple tiers. Adopted from Chang et al. (2018).

lithic inter-tier vias (MIVs). To determine the location of MIVs, the metal layers are duplicated, and the top-tier cells are assigned to the duplicated layer. After routing, the connections between two metal layers become MIVs. The authors compared the 2D and M3D designs in detail. The M3D design achieved 50.1% footprint reduction, 14.6% cell area saving, and around 30% wirelength reduction. When it comes to power consumption, M3D design shows 22.3% total power reduction in a feed-forward classification workload and 8.6% in a pseudo-training task. The total power discussed includes internal power, switching power, and leakage power. The dynamic power is computed as in the following equation:

$$\begin{aligned} P_{dyn} &= P_{INT} + P_{SW} \\ &= \alpha_{IN} \cdot I_{SC} \cdot V_{DD} \cdot f_{clk} \\ &+ \alpha_{OUT} \cdot (C_{pin} + C_{wire}) \cdot V_{DD}^2 \cdot f_{clk} \end{aligned} \tag{2.2}$$

where the first term P_{INT} counts power consumption of standard cells and memory blocks, and the second term P_{SW} represents the switching power dissipated during charging or discharging of output load capacitance of cells. The authors argue that the classification tasks rely mainly on combinational logic gates and thus it is compute-intensive. In contrary, pseudo-training needs to read/write weights and it becomes memory-intensive. Therefore, switching activity is much higher and explains the larger power consumption. The experiments conclude that M3D is a good fit for low-power DNN hardware implementation by offering performance improvement over 2D designs, especially for architecture with complex combinational logics.

Emerging beyond-CMOS devices give rise to new solutions for low-power designs. It is shown that HyperFET, a MOSFET replacement, greatly lowers the power consumption of spiking neural networks compared to the CMOS-based counterpart (Tsai et al., 2016). Similar attempts include Cellular Neural Networks on TFET (Palit et al., 2013). Liu et al. (2020) review emerging materials for next-generation computing technologies.

Chapter 3

Graph Learning-Based Arithmetic Block Identification

3.1 Motivation

Arithmetic block identification in gate-level netlists has emerged as the driving force for numerous datapath optimization or functional verification methodologies. For example, Symbolic Computer Algebra (SCA) based multiplier verification (Mahzoon et al., 2018, 2019) relies heavily on the detection of Half Adders in the multiplier netlist. It is also desired to replace a detected arithmetic block with pre-optimized logic or even new macro blocks built with more advanced technologies (Wei et al., 2015). Additionally, the demand for malicious logic detection is widely pointed out (Tehranipoor and Koushanfar, 2010; Meade et al., 2016; Li et al., 2019) to ensure circuit security and functionality in the globalization of VLSI design, manufacturing, and distribution. Besides the applications mentioned above, a technical reason behind the need for such a 'reverse engineering' approach is that most high-level components, such as function declaration and modularization, are flattened into netlists of Boolean gates by logic synthesis and technology mapping (Yu and Ciesielski, 2016). Therefore, despite sounding like '*finding a needle in a haystack*' (or in a sea of bit-level gates (Li et al., 2013)), arithmetic block identification is an essential procedure worthy of exploration.

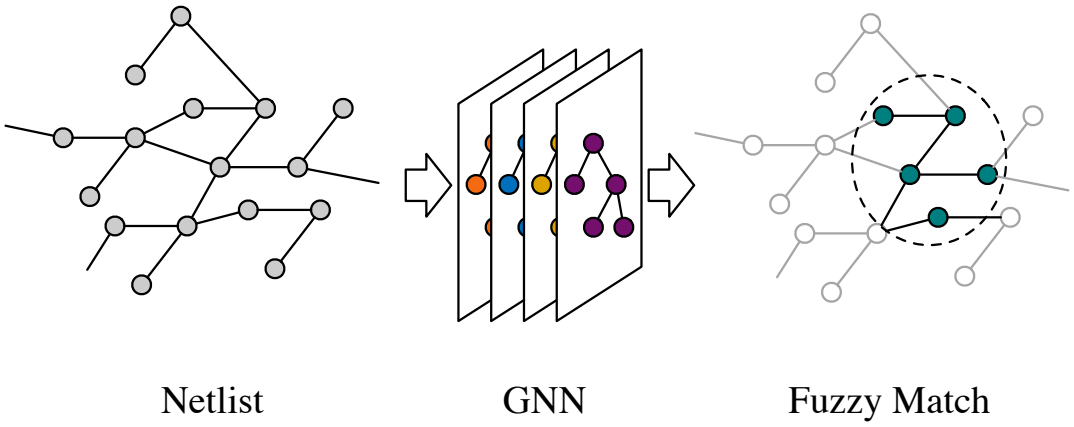


Figure 3.1: Graph learning enables netlist fuzzy matching.

As introduced in Section 2.1, classic approaches to arithmetic block identification include structural methods that focus on circuit topology and functional methods that focus on circuit function analysis. Structural methods are efficient with customized algorithms, yet they are mathematically incomplete; on the contrary, functional methods are accurate and solver-ready at the cost of ultra-long runtime. Moreover, existing machine learning driven methods are dedicated to one given unknown functional block, and thus face significant challenges when dealing with large-scale netlist design. To address the above concerns, we propose a graph learning-based arithmetic block identification framework, as briefly illustrated in Figure 3.1, that can efficiently conduct fuzzy matching on arithmetic blocks. The framework takes a large-scale netlist as input, and outputs fuzzy-matched sub-graphs as target arithmetic components. Since a netlist is often represented as a directed acyclic graph (DAG), it is motivated to utilize graph neural networks (GNNs) as the preferable fuzzy matching solution. Intuitively, GNNs can aggregate information from neighbourhoods to generate meaningful low-dimensional embeddings for each vertex for downstream tasks. However, most existing popular GNN models, such as GraphSAGE (Hamilton et al., 2017) and GIN (Xu et al., 2019), are designed for general graphs or undirected graphs. In other words, they are not well-optimized for DAGs. Therefore, we come up with a variant of GNN, *asynchronous bidirectional graph neural network* (**ABGNN**), which is customized for DAG embedding with supreme performance and high efficiency.

3.2 Problem Formulation

We first introduce the problem formulation. The *gate-level netlist* of an electric circuit consists of a list of gate-level circuit components (e.g., `AND` gates) and their interconnects. Gate-level netlists are usually generated by logic synthesis tools, which converts an abstract specification of circuit behaviour (typically at *register transfer level* (RTL)) into design implementation in terms of logic gates. Mathematically, a gate-level netlist can be naturally formulated as a directed acyclic graph, whose vertices are the circuit components and edges represent wires between them. Sometimes we emphasize a *flattened* gate-level netlist, where only primitive gates are instanced, while the design hierarchy is unknown. Within a netlist, arithmetic blocks are the building blocks that perform simple arithmetic operations, such as integer addition or subtraction. In general, our target is to discover the arithmetic blocks located in a flattened netlist. For simplicity, in this paper, we focus on identifying adders, which are one of the major arithmetic components. However, our proposed graph learning-based framework can be easily extended to other desired arithmetic blocks.

Problem 1 (Adder Identification)**.** *Given the flattened gate-level netlist of a circuit design, identify adders located in the netlist.*

3.3 Flow Overview

Before introducing algorithmic details, we briefly overview our proposed arithmetic block identification flow. Given a design netlist, we first transform it into a directed acyclic graph (DAG) representation. The DAG is fed to our designed ABGNN (introduced in Section 3.4) to generate node embeddings. The node embeddings are further used to predict arithmetic block boundary (introduced in Section 3.6.1). Then, we run a network flow-based algorithm (introduced in Section 3.5) to match the predicted input wires with the predicted outputs wires. We illustrate the overall flow in Figure 3.2.

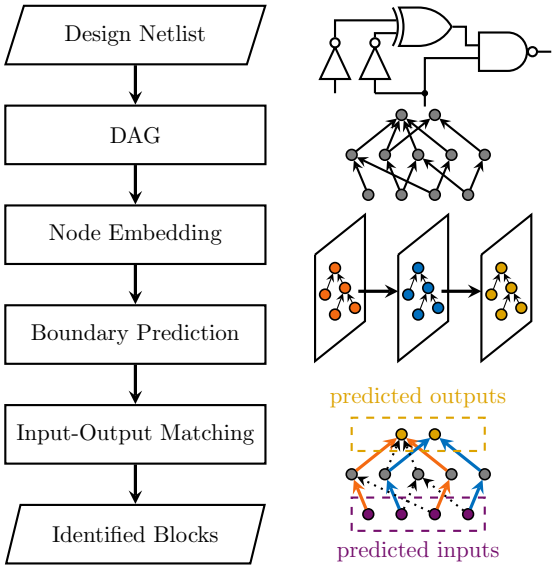


Figure 3.2: Our arithmetic block identification flow.

3.4 Designing Graph Neural Network for DAGs

Graph neural networks enable a powerful representation learning paradigm for graphs. In general, GNNs follow a neighbor aggregation (or equivalently, message passing) scheme (Xu et al., 2019): the representation vector of a node is computed by recursively aggregating and transforming representation vectors of its neighboring nodes. The above message passing scheme has achieved state-of-the-art performance on various tasks on graphs, such as node classification, link prediction, and graph classification. Nevertheless, it is still critical to customize graph neural network architecture according to the actual task to earn the best result. In this section, we discuss how do we design a novel graph neural network architecture dedicated to DAG representation learning in our adder IO prediction task.

3.4.1 General Graph Neural Network

We start with a formal introduction to general graph neural networks, partly following the notations in (Xu et al., 2019). A graph can be represented as $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V} = \{v_1, v_2, \cdots, v_n\}$ is the vertex set, and $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ is the edge set. Vertices are equipped with initial feature vectors $\mathcal{X} = \{\boldsymbol{x}_v | \forall v \in \mathcal{V}\}$. As introduced, GNNs follow a neighbor aggregation scheme. The k-th iteration

of message passing, or say the k-th layer of a GNN, can be written as follows:

$$\boldsymbol{a}_v^{(k)} = \texttt{AGGREGATE}(\{\boldsymbol{h}_u^{(k-1)} : u \in \mathcal{N}(v)\}),$$
$$\boldsymbol{h}_v^{(k)} = \texttt{COMBINE}(\boldsymbol{a}_v^{(k)}, \boldsymbol{h}_v^{(k-1)}),$$

where $\boldsymbol{h}_v^{(k)}$ is the representation vector of vertex v after k iterations, $\boldsymbol{h}_v^{(0)} = \boldsymbol{x}_v$, and $\mathcal{N}(v)$ denotes the neighbouring nodes of v. Many GNN variants with different choices of the `AGGREGATE` function and the `COMBINE` are proposed, which are crucial to the model performance. In practice, common selection of the `AGGREGATE` function include `mean` aggregators, `max` aggregators, and `sum` aggregators, usually followed by a multi-layer perceptron (MLP). As a concrete example, *GraphSAGE* (Hamilton et al., 2017), one of the dominantly used architectures, aggregates neighborhood information in the following way:

$$\boldsymbol{h}_v^{(k)} = \sigma(\boldsymbol{W} \cdot \texttt{MEAN}(\{\boldsymbol{h}_v^{(k-1)}\} \cup \{\boldsymbol{h}_u^{k-1} : u \in \mathcal{N}(v)\})),$$

where σ is an activation function (e.g. `sigmoid`). This is also a rough, linear approximation of a localized spectral convolution.

3.4.2 Bidirectional Graph Neural Network

We now come to the first keyword, '*Directed*', of '*Directed Acyclic Graph*'. Directed graphs assign each edge a direction, which naturally captures various real-life relations. In our netlist, the edge direction indicates the current flow direction, or say the execution order of the circuit. Therefore, modeling a netlist with a directed graph is intrinsic and significant.

However, most existing GNN models assume to work for undirected graphs. One historical reason is that, earlier spectral GNN models (Bruna et al., 2014; Defferrard et al., 2016; Kipf and Welling, 2017), built upon the analogy to Convolutional Neural Networks (CNNs), define the *graph convolution* as the multiplication of a signal $\boldsymbol{x} \in \mathbb{R}^N$ with a filter $\boldsymbol{g}_\theta = \text{diag}(\boldsymbol{\theta})$ parameterized by $\boldsymbol{\theta} \in \mathbb{R}^N$ in the Fourier domain, namely:

$$\boldsymbol{g}_\theta \star \boldsymbol{x} = \boldsymbol{U}\boldsymbol{g}_\theta\boldsymbol{U}^\top\boldsymbol{x},$$

where $\boldsymbol{U}$ is the matrix of eigenvectors of the normalized graph Laplacian $\boldsymbol{L} = \boldsymbol{I}_N - \boldsymbol{D}^{-\frac{1}{2}}\boldsymbol{A}\boldsymbol{D}^{-\frac{1}{2}} =$

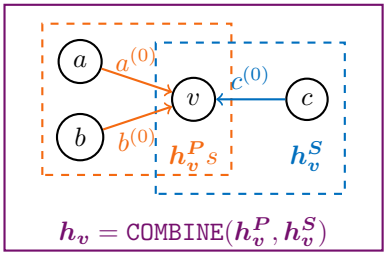


Figure 3.3: Bidirectional information aggregation for the vertex v. We train two GNNs to aggregate information from the fanin cone ($\boldsymbol{h}_v^P$, in orange) and the fanout cone ($\boldsymbol{h}_v^S$, in blue) respectively. The final embedding ($\boldsymbol{h}_v$, in purple) is given by the combination of both representation vectors.

$\boldsymbol{U}\boldsymbol{\Lambda}\boldsymbol{U}^\top$. In this definition, $\boldsymbol{U}^\top\boldsymbol{x}$ is considered the *graph Fourier transform* of $\boldsymbol{x}$, which relies on the fact that the (real symmetric) normalized graph Laplacian $\boldsymbol{L}$ admits an eigendecomposition. Unfortunately, we do not directly have this property for a directed graph. One straightforward way is to relax the directed graph to an undirected graph by symmetrizing its adjacency matrix, but this inevitably results in information loss.

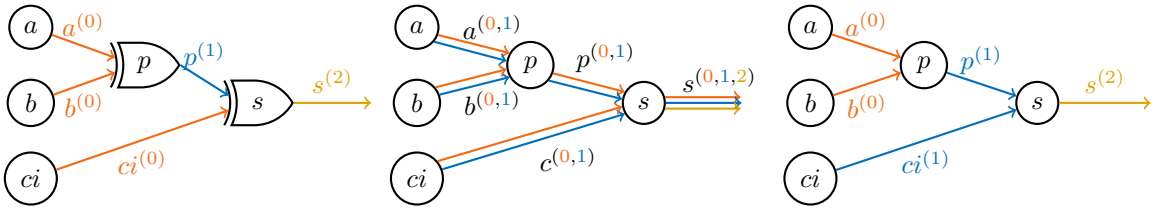
Our designed bidirectional GNN is greatly motivated by the design of heterogeneous GNNs (Zhang et al., 2019b; Wang et al., 2019a). As discussed in (Zhang et al., 2019b), one of the challenges in designing heterogeneous GNN is 'how to aggregate feature information of heterogeneous neighbors by considering the impacts of different node types'. In arithmetic block identification, the role of a gate depends on both its fanin cone and its fanout cone. It is therefore necessary to combine information from both directions to generate representative node embeddings. Hereafter, we denote the *transpose graph* as $\mathcal{G}^\top$, which contains a directed edge (u, v) if and only if $\mathcal{G}$ contains the reversed edge (v, u).

To encode the edge directions, each vertex **only aggregates information from its *predecessors***. In other words, information flows from x to y if there is an edge (x, y). We train two GNNs, one for $\mathcal{G}$ and one for the transpose graph $\mathcal{G}^\top$, to generate two embedding vectors $\boldsymbol{h}_v^P$ and $\boldsymbol{h}_v^S$ for each vertex that aggregate information from the predecessors (i.e., fanin cone) and the successors (i.e., fanout cone) respectively. Thus, the final embedding of each vertex is given by the combination of both $\boldsymbol{h}_v^P$ and $\boldsymbol{h}_v^S$:

$$\boldsymbol{h}_v = \texttt{COMBINE}(\boldsymbol{h}_v^P, \boldsymbol{h}_v^S) \tag{3.1}$$

The placeholder `COMBINE` can be any common reduction function such as `mean`, `max`, or `sum`. In

practice, we simply concatenate the two vectors for the final embedding. Figure 3.3 demonstrates the bidirectional information aggregation scheme for a vertex.



(a) Distributed Logic Simulation (b) Synchronous GNN (c) Asynchronous GNN

	(a) Logic Simulation					(b) Synchronous					(c) Asynchronous				
	a	b	ci	p	s	a	b	ci	p	s	a	b	ci	p	s
T = 0	✓	✓	✓			✓	✓	✓	✓	✓	✓	✓			
T = 1				✓		✓	✓	✓	✓	✓			✓	✓	
T = 2					✓					✓					✓

(d) Message passing comparison. A checkmark indicates that the node sends out message at the timestamp.

Figure 3.4: A comparison between (a) distributed logic simulation, (b) synchronous GNN message passing, and (c) asynchronous GNN message passing. The table in (d) lists messages sent out by nodes at each timestamp. Asynchronous GNNs are more efficient than synchronous GNNs.

3.4.3 Asynchronous Graph Neural Network

We move to the second keyword, 'Acyclic', of 'Directed Acyclic Graph'. Acyclic graphs contain no cycles. That is, if we start from any vertex v, walking through the graph following the edge directions, we will never come back to v. Although this property may sound irrelevant to GNN design, we show that it is possible to improve the efficiency of GNN utilizing the acyclic property.

Let's make an analogy to event-driven logic simulation, taking the Chandy-Misra-Bryant (CMB) distributed-time algorithm (Chandy and Misra, 1981) as an example. To enable parallel logic simulation with the CMB algorithm, circuit elements communicate with each other using timestamped messages, and different elements may consume events at distinct simulation times concurrently. Conceptually, each element consumes timestamped event messages on its inputs; whenever all inputs are ready, it advances its local time and possibly sends out new time stamped event messages on its output. Figure 3.4a illustrates the event message scheme assuming a unit delay for each gate. The

original CMB algorithm is regarded as '*an approach to carry out asynchronous, distributed simulation on multiprocessor message-passing architectures*' (Chandy and Misra, 1981). On the contrary, general GNNs work in a *synchronous* way. In synchronous message passing, all messages flow on edges simultaneously in each iteration, such that every vertex receives messages and updates its representation on every iteration, causing great computational demand, as shown in Figure 3.4b.

Motivated by the CMB algorithm and the acyclic nature of the netlist, we propose an asynchronous GNN architecture, resembling the asynchronous message-passing scheme for logic simulation. To embed a target vertex v, consider its fanin cone rooted at v. The message passing process starts from the leaf nodes of the cone, through the cone, and all the way up to v. At each 'timestamp', only the vertices receive messages at the previous timestamp pass the message to their direct successors. Figure 3.4c shows an example to embed node s using such an asynchronous GNN. In iteration 0, only nodes a and b send out their messages to p, while in iteration 1, node p and node ci send out their messages to s. The table in Figure 3.4d lists are the messages sent out by nodes at each timestamp. We can see that asynchronous GNN executes as efficiently as logic simulation while being much more efficient than synchronous message passing.

Formally, for a target vertex v, the aggregation scheme of the k-th iteration of a depth-Δ asynchronous GNN can be described as follows:

$$\begin{aligned} \boldsymbol{a}^{(k)}_{\{\mathbf{i}:\mathcal{D}(\mathbf{i},\mathbf{v})=\boldsymbol{\Delta}-\mathbf{k}\}} &= \texttt{AGGREGATE}(\{\boldsymbol{h}_u^{(k-1)} : u \in \mathcal{N}(i)\}), \\ \boldsymbol{h}^{(k)}_{\{\mathbf{i}:\mathcal{D}(\mathbf{i},\mathbf{v})=\boldsymbol{\Delta}-\mathbf{k}\}} &= \texttt{COMBINE}(\boldsymbol{a}_i^{(k)}, \boldsymbol{h}_i^{(\mathbf{0})}), \end{aligned} \tag{3.2}$$

where $\mathcal{D}(i, v)$ is the distance between vertices i and v in the graph, and $\boldsymbol{h}_i^{(0)}$ is the initial feature of vertex i. The boldface indices emphasize the difference compared with a general GNN. In other words, in the k-th iteration of a depth-Δ asynchronous GNN, only those **vertices whose distance to the root v is $\Delta - k$ aggregates information** from its predecessors. Then, the aggregated embedding is **combined with their initial features** as the representation vector. In this way, unlike synchronous GNNs, messages are passed through each edge exactly only once (in the embedding of each node), which saves lots of computational efforts.

3.4.4 Putting It All Together

In previous subsections, we propose two special GNN architectural structures, namely *bidirectional* and *asynchronous*, according to the directed and acyclic properties of the target graph (DAG), respectively. The two structures are orthogonal, so that we can combine them in our final GNN architecture, asynchronous bidirectional graph neural network (ABGNN). We evaluate the performance of ABGNN in Section 3.7.5.

3.4.5 Related Works for DAG Embedding

There exist several works that aim to design graph learning models for DAGs. DAGNN (Thost and Chen, 2021) constructs a multi-layer network to generate an embedding for the whole DAG. The network is driven by the partial order induced by the DAG. However, their model is still computationally expensive since they use an iterative message passing scheme. Moreover, they use Gated Recurrent Units (GRUs) as the `combine` operator, further increasing the inference time. D-VAE (Zhang et al., 2019a) proposes an asynchronous message passing scheme to encode the computations on DAGs, which is the most similar work to ours. However, ABGNN differs from D-VAE since we focus on local structures and thus the generation of node-level embeddings, whereas D-VAE encodes information of the whole (computation) graph.

3.5 Input-output Matching

Our proposed graph learning framework identifies the boundary of arithmetic blocks. In particular, the model predicts the input wires and the output wires of arithmetic blocks. What if we want further to match the input bits with the corresponding output bits? In this section, we propose to use a network-flow-based algorithm to extract the datapaths within an arithmetic block. The problem of datapath extraction has gained great attention since it is believed datapath-aware physical synthesis may achieve higher performance. Readers are referred to He et al. (2021) for a survey for datapath extraction approaches and datapath-driven placement methodologies. For now, we illustrate the feasibility of the network-flow approach for adder IO matching, and leave the other possible solutions for future work, since it is beyond the main scope of this paper.

The datapath extraction problem for an adder is defined as follows: given an (unordered) adder input set $S = A \cup B$ where $A = \{a_0, \cdots, a_{n-1}\}, B = \{b_0, \cdots, b_{n-1}\}$ and an (unordered) adder output set $T = \{t_0, t_1, \cdots, t_{n-1}\}$ such that $T[n-1:0] = A[n-1:0] + B[n-1:0]$, identify $2n$ datapaths from S and T such that **1)** all wires in S and T are covered, and **2)** each datapath starts from a_i or b_i ends at t_i. Inspired by Xiang et al. (2013), we formulate the problem as a *maximum flow problem*. We add a pseudo *source* node S^* and a pseudo *sink* node T^* in the graph, and edges from S^* to every node in S, as well as every node in T to T^*. The newly added edges from S^* to nodes in S are assigned unit capacity, while the rest edges are assigned capacity of 2. Then we run a *maximum-flow algorithm* to find the routes between S and T.

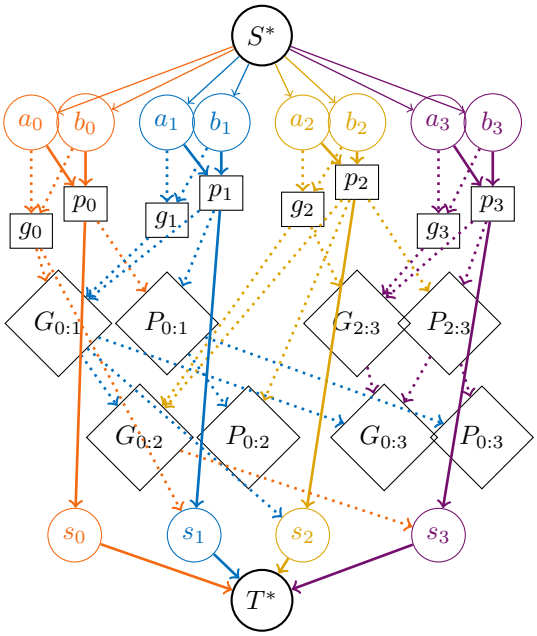


Figure 3.5: A Brent-Kung adder example to demonstrate that the unique maximum flow matches inputs and outputs correctly. We analyze the flows in the order of orange, blue, yellow, to purple. Solid lines are charged with flows, and dotted lines are banned due to flow capacity constraints.

We illustrate the feasibility of the maximum-flow algorithm by taking the *Brent-Kung adder* (Brent and Kung, 1982) as an example. The maximal flow network is shown in Figure 3.5. We analyze the flows in the order of orange, blue, yellow, to purple. The solid edges are charged with flows, while the dotted edges are banned due to flow capacity constraints. In fact, this is the **unique maximum flow solution** of the flow network. To charge the flow from s_0 to T^* (with flow value

2), the only possible route is to pass from p_0 to s_0, which occupies the edges from S^* to a_0, b_0 and up to p_0. These edges are marked with solid orange lines, and meanwhile, the edges banned due to capacity constraints are marked with dotted orange lines, such as other edges starting from a_0 and b_0. Immediately, we observe that there is no route to charge the node g_0, which further indicates that there is no flow from g_0 to s_1, leaving (p_1, s_1) the only possible route to charge the flow from s_1 to T^*. Then we can do the same analysis for s_1 to T^* with blue lines, s_2 to T^* with golden lines, and s_3 to T^* with purple lines. Finally we will see the uniqueness of the maximum flow, and the matching is done.

How to better orchestra the network flow approach with a fuzzy matching framework deserves more discussions. Although our theoretical analysis finds the maximum flow solution unique, there is no such guarantee if we are given fuzzy, imperfect predictions of adder boundaries. However, we observe that besides input-output matching, the network flow approach also acts as a filter to remove some false alarms. In other words, if the maximum flow does not flow over some predicted input/output node, then the node is actually unlikely to be a boundary node of an adder. Inspired by so, we propose to run the maximum flow algorithm in both directions (inputs to outputs and outputs to inputs), so that the maximum forward flow (inputs to outputs) filters out false alarms of predicted outputs, and vice versa. To retain high sensitivity, we also add the siblings of predicted input (output) nodes in the forward (backward) runs, so that we are confident enough in the filter. This strategy indeed improves prediction precision with almost no sensitivity loss in our experiments.

3.6 Other Algorithm Details

This section describes other important algorithm details of our arithmetic block identification flow, including discussions on the learning problem formulation, and the strategies to deal with the data imbalance issue.

3.6.1 Machine Learning Problem Formulation

As we introduced, we utilize a customized GNN for netlist representation learning. However, how to learn the parameters in the GNN model remains to be considered. Essentially, the arithmetic

block identification problem is to 'detect' instances of target semantic objects in the graph, which sounds like a graph version of the *object detection* task in computer vision. Despite the intuitive descriptions, solving such problems is still very challenging for the community due to **1)** the NP-complete nature of the problem and **2)** the requirement to consider graph topology, node features, and/or edge features at once.

Given that, we propose to formulate a *node classification* problem to circumvent the hard-to-solve graph detection problem. Specifically, the target of our neural model is to **predict boundary (input wires and output wires) of arithmetic blocks**. Another possible problem formulation is to predict the region of arithmetic blocks, which is abandoned after comparison. Note that a wire can be both an input to one arithmetic block and an output from another arithmetic block (consider the two expressions $c = a + b$ and $e = c + d$, where c is the output of the first adder and the input of the second adder). Therefore, we use *input prediction* and *output prediction* to refer to the two independent binary classification tasks for boundary prediction. We use an MLP with binary cross-entropy loss to consume the representation vectors generated by GNN and carry out the prediction.

3.6.2 Dealing with Data Imbalance

The data imbalance issue refers to the phenomenon that some classes (*majority*) have a significantly higher number of examples in the training set than other classes (*minority*). It is a common problem in real-life applications from various domains, which has been established to have a significant detrimental effect on training classifiers in terms of both training convergence and generalization ability (Buda et al., 2018). For example, it is observed that the model would easily lean towards majority classes (Kang et al., 2020), making some standard metrics like *accuracy* invalid (since they may cause misinterpretation of data). We refer readers to (Ali et al., 2015) for a comprehensive review. In our dataset, the ratio of negative nodes to positive nodes is around $100 : 1$, which is indeed highly imbalanced.

Methods to address data imbalance can be divided into two categories, namely data-level methods and algorithm-level methods. Data-level methods aim to alter the distribution of the training dataset so that standard algorithms for balanced data can work well. On the other hand, algorithm-level

methods keep the training dataset unchanged and adjust the training/inference algorithm. We now introduce three techniques we adopt in our training.

Oversampling

Oversampling is one of the most popular data-level methods used in machine learning. We adopt the basic version of it, called random minority oversampling, which simply **replicates randomly selected samples from minority classes** (Buda et al., 2018). Some more advanced oversampling methods (e.g., SMOTE (Chawla et al., 2002)) have also been proposed, which we leave for possible future work.

Cost Sensitive Learning

Cost sensitive learning (Elkan, 2001) assigns different penalties to different misclassification errors. Mathematically, if C_{ij} refers to the cost for predicting class j when the actual class is i, the optimal prediction for an example x is given by

$$\operatorname*{argmin}_i \sum_j p(j|x)C_{ij},$$

where $p(j|x)$ is the estimated probability of example x being in class j.

We encode cost sensitive learning into the loss function. Let the total loss $\mathcal{L}$ be decoupled into two parts, namely the loss on the positive samples ($\mathcal{L}_{pos}$) and the loss on the negative samples ($\mathcal{L}_{neg}$). Since negative samples are the majority, we **assign a penalty weight α ($\alpha < 1$) to the negative loss**, so that the contribution of negative nodes to the total loss function is reduced, which compensates the imbalance between sample classes. The weighted loss function can be formulated explicitly as:

$$\mathcal{L} = (\mathcal{L}_{pos} + \alpha\mathcal{L}_{neg})/N, \tag{3.3}$$

where N is total number of samples.

3.7 Experiments

3.7.1 Setup

We develop the graph object detection framework with DGL (Wang et al., 2019c), a graph learning library, which is based on PyTorch (Paszke et al., 2017) for tensor manipulations. The network flow algorithm (viz. *Edmonds–Karp*) is implemented with networkx (Hagberg et al., 2008). We also refer to the EPFL logic synthesis libraries (Soeken et al., 2018) when we reimplement the baseline methods. Graph neural networks are trained on a Linux machine with 48 Intel Xeon Silver 4212 cores (2.20GHz), 1 GeForce RTX 2080 Ti graphics card, and 32 GB main memory. Training details are discussed in subsequent sections.

3.7.2 Dataset

The dataset we use comes from open-source RISC-V CPU designs (Amid et al., 2020), including *Rocket* (Asanovic et al., 2016), a 5-stage in-order scalar core, and Berkeley Out-of-Order (*BOOM*) Core (Asanovic et al., 2016), an out-of-order superscalar RV64G core. Since *BOOM* is more complicated (around 5x larger than *Rocket*), we use it as the training set, while leaving *Rocket* as the testing set.

The netlists are automatically generated from Chisel, which is further synthesized with Synopsys Design Compiler targeting the SAED 32/28nm Digital Standard Cell Library. For each design, we synthesize a set of netlists using various design constraints, so that different adder designs could be generated by DC.

Statistics of the generated netlists are listed in Table 3.1. In fact, there are other related constraints that could be specified, such as the radix of the prefix structure in adders or some timing constraints. In our experiments, we observe very similar outcomes as we adjust this set of constraints, so we simply omit them for simplicity.

3.7.3 Baselines

We reimplemented several representative literature works (Subramanyan et al., 2013a; Wei et al., 2015; Fayyazi et al., 2019) as the baseline methods for comparison. These works have covered

Architecture	*Rocket*		*BOOM*	
	#gates	#wires	#gates	#wires
Brent-Kung	24340	58124	139526	366280
Cond-sum	24737	57708	138358	360455
Hybrid	25491	60287	141319	369622
Kogge-Stone	24540	57726	139005	361962
Ling	26179	62864	143903	378354
Sklansky	25208	59567	141093	369774

Table 3.1: Statistics of the dataset. We use *BOOM* as the training set as it is more complicated, leaving *Rocket* as the testing set. We synthesize a set of netlists for each design by specifying different adder architectures in Design Compiler.

structural methods, functional methods, as well as machine learning methods in their proposed solutions. Subramanyan et al. (2013a) first enumerates all cuts[1] and groups them into permutation-independent equivalence classes, which are then aggregated into candidate *words* based on *common support* or *signal propagation*. The candidate words are further propagated in the graph to form new words based on neighboring gate types. We optimistically estimate the performance upper bound of the algorithm without running *symbolic simulation* and *equivalence checking*, but simply include all the potential words instead. Wei et al. (2015) builds `xor` trees, identifies carry-out signals, and constructs `xor`-forests based on the connection hierarchy. Fayyazi et al. (2019) proposes to represent circuit topology using *level-dependent decaying sum* (LDDS) *existence vector* (EV), which basically marks the gate types that appeared in a local subgraph and assigns distance-based penalty weights. We follow the LDDS-EV construction, expect that we clip all large values in the EV to 64, and add a batch normalization layer in the neural network to stabilize training. We also apply the oversampling technique by using a weighted random sampler during training. Since this method was originally evaluated for circuit classification, we adapt the method to our problem formulation and our proposed flow.

[1] Subramanyan et al. (2013a) suggested enumerating 6-feasible cuts, but our reported results are based on 5-feasible cut enumeration because it yields almost the same performance with much shorter (0.01×) runtime.

3.7.4 Overall Comparison

We first compare the performance between our proposed method and all baseline approaches (Subramanyan et al., 2013a; Wei et al., 2015; Fayyazi et al., 2019) as introduced in Section 3.7.3. The results are listed in Table 3.5.

Our proposed arithmetic block identification method greatly outperforms prior works on all the testcases, averaged 95.1% and 93.7% sensitivity in input and output boundary identification, respectively. It is also the fastest method even though we run a maximum flow algorithm for input-output matching. The other machine learning approach (Fayyazi et al., 2019) achieves the second-best performance (83.4% and 76.1% sensitivity), but its precision (around 0.35 on average) is in fact much lower than ours (over 0.94 on average). Nevertheless, it still confirms the good adaptability of deep learning methods and the effectiveness of the oversampling strategy for imbalanced datasets. Subramanyan et al. (2013a) is able to cover lots of words composed of replicated functional bitslices, and therefore achieves acceptable sensitivity (81.9% and 56.5%), at the cost of much higher runtime (37.5× over ours). Wei et al. (2015) is stable for the more regular architectures (Cond-sum, *Kogge-Stone*), but does not perform well given complicated or highly optimized structures (Hybrid, *Ling*), resulting in unsatisfactory average sensitivity (49.9% and 43.5%).

3.7.5 Evaluation of ABGNN

We conducted comprehensive experiments to evaluate our proposed graph neural network architecture and demonstrate its outstanding capability in DAG representation learning. We set the fanin depth and fanout depth of ABGNN to 1 and 5 respectively for input boundary prediction, and $(2, 2)$ for output boundary prediction.

Table 3.2: Comparison between asynchronous and synchronous GNNs. Asynchronous GNNs reduce inference time without performance degradation.

Task	Model	Recall	F_1-score	Runtime (ms)
Input	asynchronous	**0.951±0.000**	**0.956±0.000**	**122.1**
	synchronous	0.943±0.003	0.951±0.002	152.2
Output	asynchronous	**0.937±0.015**	**0.940±0.012**	**77.6**
	synchronous	0.933±0.012	0.937±0.009	94.6

Comparison with State-of-the-Art GNNs We evaluate our proposed ABGNN with several state-of-the-art Graph Neural Networks, including GAT (Velickovic et al., 2018), GIN (Xu et al., 2019), and GraphSAGE (Hamilton et al., 2017), on the *Rocket* dataset. Our model achieves the best performance on all the cases with much higher recall and F1 scores, showing its superiority on DAG representation learning. In some complex cases (e.g., input prediction in the Brent-Kung case), our model outperforms other models by 5%–9% for the F1 score. On average, our model achieves 2.8%-5.0% recall gain and 3.3%-9.5% F1 score gain in input identification (Table 3.6), as well as 1.9%–6.2% recall gain, and 2.6%–7.0% F1 score gain in output identification (Table 3.7).

Effect of asynchronous message passing We conducted experiments to verify the effect of the asynchronous message passing scheme by comparing it with synchronous GNNs, while leaving other hyper-parameters the same, including the number of layers, oversampling rate, etc. Table 3.2 shows that compared with synchronous GNNs, asynchronous GNNs reduce inference time by 19.8% and 18.0% respectively for input and output boundary identification, without any performance degradation. Here the runtime refers to the inference time of GNN, namely the time the model takes to generate node representations. We want to emphasize that the efficiency will likely improve as the model depth increases (confirmed by our preliminary experiments), and thus the asynchronous GNN might work even better for more complicated tasks.

Effect of bidirectional information aggregation We also carry out experiments to see the effects of bidirectional information aggregation. We build unidirectional models by reducing fanin depth to 0 for input boundary identification and fanout depth to 0 for output identification. As shown in Table 3.3, bidirectional information aggregation improves 4.6% recall and 11.1% F_1-score for the output model, as well as 1.8% recall and 2.1% F_1-score for the input model. The performance gain indicates that information from a single direction is not sufficient to identify thhe input/output boundary of an adder, and therefore combining representations learned from both directions is indeed necessary.

Table 3.3: Comparison between bidirectional and unidirectional GNNs. Bidirectional GNNs outperform unidirectional GNNs, confirming the effectiveness of bidirectional informatrion aggregation.

Task	Model	Recall	F_1-score
Input	bidirectional	**0.951±0.000**	**0.956±0.000**
	unidirectional	0.933±0.002	0.935±0.002
Output	bidirectional	**0.937±0.015**	**0.940±0.012**
	unidirectional	0.891±0.001	0.829±0.011

Table 3.4: Performance of ABGNN on region detection and boundary identification. The performance of boundary identification is much better.

Formulation	Task	Recall	F_1-score
boundary	input	0.951 ± 0.000	0.956 ± 0.000
	output	0.937 ± 0.015	0.940 ± 0.012
region	internal	0.762 ± 0.022	0.737 ± 0.025

3.7.6 Other Analysis

Region Detection vs. Boundary Identification As mentioned in Section 3.6.1, there exist two different problem formulations for adder identification, namely region detection and boundary identification. Note that we adopt the latter one in our final solution. For region detection, the idea is to assign a positive label to all the nodes within the adder (including I/O wires). Table 3.4 shows the performance of ABGNN targeting region detection and boundary identification. It can be seen that ABGNN performs far better on boundary identification than on region detection. The result is actually within our expectations since internal nodes are not as distinguishable as boundary nodes.

Effect of Oversampling Recall that we utilize the oversampling strategy to deal with the data imbalance issue. Figure 3.6 demonstrates the testing performance curves (recall, F_1-score) during the 100 training epochs using four different oversampling rates (1, 5, 10, 20). Without oversampling (the orange curve), training is quite unstable and converges slower (and finally to a worse model), confirming the effectiveness of oversampling. In our evaluation, we use an oversampling rate of 5.

Table 3.5: Overall performance comparison on the test set (the *Rocket* core). Best results are emphasized with **boldface**, and second-best results are colored in blue. Our proposed arithmetic block identification method greatly improves boundary recognition performance compared with previous works. It also runs the fastest among all the methods.

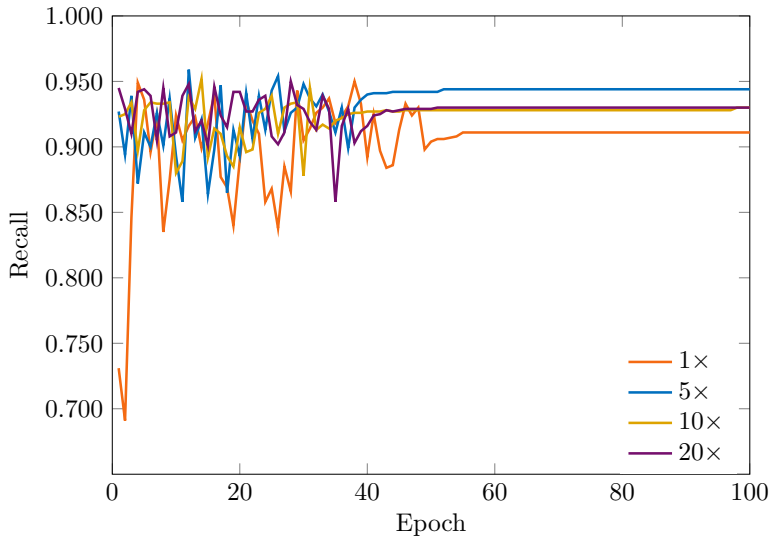
Case	Subramanyan et al. (2013a)			Wei et al. (2015)			Fayyazi et al. (2019)			Ours		
	Input	Ouput	Runtime(s)	Input	Ouput	Runtime(s)	Input	Ouput	Runtime(s)	Input	Ouput	Runtime(s)
Brent Kung	0.826	0.672	302.0	0.554	0.493	13.4	0.875±0.022	0.820±0.013	11.6±3.9	**0.950±0.000**	**0.954±0.020**	**10.2±1.8**
Cond-sum	0.825	0.598	380.6	0.770	0.787	14.6	0.808±0.013	0.744±0.020	13.0±3.7	**0.949±0.000**	**0.866±0.014**	**10.9±0.6**
Hybrid	0.815	0.389	597.2	0.179	0.042	15.4	0.820±0.032	0.699±0.026	15.1±5.1	**0.947±0.000**	**0.957±0.018**	**12.0±0.7**
Kogge-Stone	0.823	0.648	525.2	0.755	0.783	15.8	0.763±0.015	0.810±0.011	13.2±3.5	**0.944±0.000**	**0.961±0.010**	**11.0±0.9**
Ling	0.803	0.456	315.6	0.249	0.022	16.5	0.874±0.013	0.653±0.074	16.3±5.5	**0.954±0.000**	**0.944±0.015**	**13.2±0.9**
Sklansky	0.823	0.626	467.4	0.484	0.483	14.7	0.864±0.017	0.845±0.017	14.1±3.7	**0.960±0.000**	**0.938±0.010**	**11.9±0.5**
Average	0.819	0.565	431.3	0.499	0.435	15.1	0.834±0.019	0.761±0.027	13.9±4.2	**0.951±0.000**	**0.937±0.015**	**11.5±0.9**

Table 3.6: Performance of different models recognizing input boundaries of adders on the test dataset (the *Rocket* core). Best results are emphasized with **boldface**, and second-best results are colored in blue. Our proposed ABGNN outperforms other models in all the test cases.

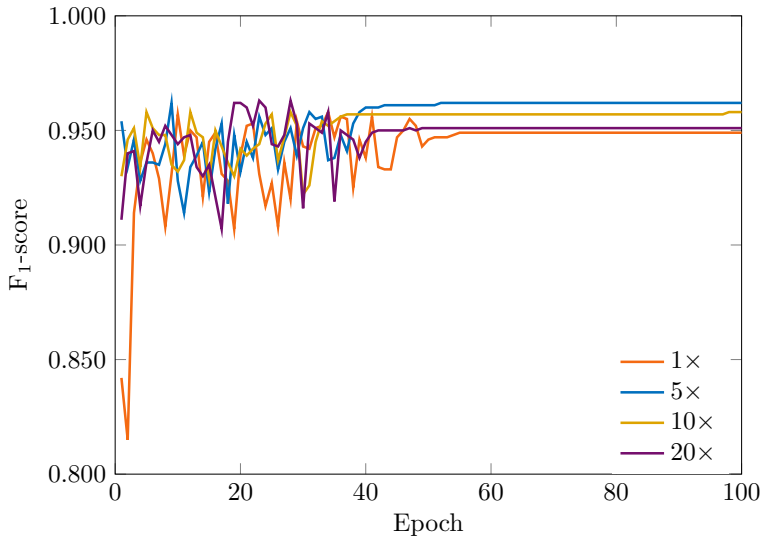
Case	GAT (Velickovic et al., 2018)		GIN (Xu et al., 2019)		GraphSage (Hamilton et al., 2017)		ABGNN (Ours)	
	Recall	F_1-score	Recall	F_1-score	Recall	F_1-score	Recall	F_1-score
Brent Kung	0.906±0.009	0.910±0.015	0.809±0.019	0.873±0.018	0.915±0.021	0.915±0.017	**0.950±0.000**	**0.964±0.000**
Cond-sum	0.882±0.022	0.885±0.018	0.875±0.024	0.890±0.021	0.935±0.015	0.931±0.016	**0.949±0.000**	**0.951±0.000**
Hybrid	0.903±0.021	0.890±0.022	0.930±0.017	0.937±0.016	0.930±0.008	0.928±0.005	**0.947±0.000**	**0.949±0.000**
Kogge-Stone	0.918±0.016	0.887±0.019	0.925±0.015	0.917±0.015	0.940±0.005	0.920±0.008	**0.944±0.000**	**0.954±0.000**
Ling	0.915±0.016	0.881±0.019	0.930±0.009	0.925±0.005	0.950±0.004	0.941±0.011	**0.954±0.000**	**0.963±0.000**
Sklansky	0.901±0.021	0.895±0.022	0.935±0.009	0.938±0.011	0.928±0.012	0.930±0.006	**0.960±0.000**	**0.955±0.000**
Average	0.904±0.017	0.891±0.018	0.901±0.016	0.913±0.013	0.933±0.011	0.923±0.010	**0.951±0.000**	**0.956±0.000**

Table 3.7: Performance of different models recognizing output boundaries of adders in the test dataset (the *Rocket* core). Best results are emphasized with **boldface**, and second-best results are colored in blue. Our proposed ABGNN outperforms other models in all the test cases.

Case	GAT (Velickovic et al., 2018)		GIN (Xu et al., 2019)		GraphSage (Hamilton et al., 2017)		ABGNN (Ours)	
	Recall	F_1-score	Recall	F_1-score	Recall	F_1-score	Recall	F_1-score
Brent-Kung	0.845±0.019	0.870±0.020	0.906±0.011	0.921±0.012	0.925±0.025	0.923±0.016	**0.953±0.019**	**0.950±0.016**
Cond-sum	0.785±0.015	0.798±0.013	0.835±0.017	0.863±0.007	0.863±0.022	0.880±0.011	**0.866±0.015**	**0.905±0.009**
Hybrid	0.940±0.024	0.897±0.021	0.911±0.021	0.875±0.008	0.93±0.016	0.912±0.016	**0.955±0.019**	**0.939±0.016**
Kogge-Stone	0.878±0.019	0.889±0.021	0.941±0.015	0.915±0.014	0.945±0.014	0.942±0.016	**0.965±0.011**	**0.955±0.015**
Ling	0.935±0.009	0.909±0.013	0.916±0.004	0.912±0.011	0.912±0.016	0.911±0.01	**0.945±0.015**	**0.948±0.007**
Sklansky	0.865±0.016	0.855±0.016	0.898±0.019	0.894±0.011	0.932±0.013	0.919±0.018	**0.938±0.012**	**0.943±0.008**
Average	0.875±0.017	0.870±0.017	0.901±0.016	0.897±0.01	0.918±0.017	0.914±0.015	**0.937±0.015**	**0.940±0.012**



(a) Curves of recall values on the test set during training.



(b) Curves of F_1-scores on the test set during training.

Figure 3.6: Test performance curves during training using different oversampling rates. Curve 1× implies no oversampling. Oversampling stabilizes training and improves model performance.

3.8 Discussion: Fast Cut Enumeration

3.8.1 Motivation

Cut enumeration enumerates bounded size cuts of all vertices in a graph, which is an important procedure in various algorithms. For example, in LUT-based FPGA technology mapping (Chen and Cong, 2004; Mishchenko et al., 2007a), cut enumeration effectively finds all possible packing strategies rooted at a node. In logic rewriting, cuts rooted at nodes are iteratively selected and replaced by smaller pre-computed subgraphs (Mishchenko et al., 2006; Li and Dubrova, 2011). In reverse engineering, cut enumeration is usually utilized to identify circuit components that match the target functionality (Subramanyan et al., 2013c; Wei et al., 2015). In theory, the number of K-bounded size cuts can be as large as $O(n^K)$, where n is the number of nodes in the circuit. Therefore, it is pointed out in several work that cut enumeration might become the runtime bottleneck of their applications (Ling et al., 2007; Possan et al., 2018).

Most existing cut enumeration algorithms follow a dynamic programming framework: the enumeration is performed from primary input nodes (PIs) to primary output nodes (POs), and the cuts of a node is generated by manipulating the cuts of its fan-in nodes. Depending on the scheme of cut generation, the manipulation can be either bottom-up or top-down. These methods usually suffer from severe redundancy issue during computation. As an example, it is reported (Takata and Matsunaga, 2009) that the number of cut products, most of which are invalid, can be about 5200 times larger than that of cuts. The above facts highlight the demand for new cut enumeration algorithms that reduce redundancy for better runtime.

In this section, we propose a new cut enumeration algorithm that combines the strengths of both bottom-up and top-down manners. Specifically, we store the cuts of a node in a tree structure, which we called cut-tree in this paper. Unlike cut-sets in usual practice, cut-trees preserve the topological relation between cuts, and this enables a few pruning strategies including subtree pruning and path compression, as we will introduce in Section 3.8.3. The central intuition is that the bottom-up scheme keeps more information about its fanin cuts, while the enforced expansion order in the top-down flow gets rid of some redundant computation. Our key idea is thus to enable 'order' in cut merge, so that the advantages of both schemes can be assembled. We prove in Section 3.8.3 that

the workload of our proposed algorithm is lower than that of both the standard bottom-up and top-down approaches for cut enumeration.

3.8.2 Preliminaries and Related Work

Logic Representation

A combinational circuit (interchangeably, boolean network) can be regarded as a Directed Acyclic Graph (DAG), where each vertex $v \in V$ represents a logic gate and the edges E represent the connecting wires between the gates. Nodes driving a node are called the *fanin* of the node, and nodes driven by a node are called the *fanout* of the node. A node with 0 fanin is called a *Primary Input* (PI), and a node with 0 fanout is called a *Primary Output* (PO). If there is a path from n_i to n_j, we say n_i is a *transitive fanin* (TFI) of n_j, and n_j is a *transitive fanout* (TFO) of n_i.

Every boolean network can be represented as a functional-equivalent *And-Inverted Graph* (AIG), composed of 2-input AND gates and inverters. A common practice is to consider an inverter as a complemented edge, which indicates the inversion of the incoming signal. Without loss of generality, we assume the network to be an AIG in the following discussions.

Cut

A cut of a node n is a set of nodes $\mathcal{C}$ in its transitive fanin such that every path from a PI to n includes a node in $\mathcal{C}$. A cut is irredundant (undominated) if no subset of it is a cut. A *K-cut* is an irredundant cut of size K, and a *K-feasible cut* is an irredundant cut whose size is no more than K. A *trivial cut* is a cut of size 1.

Cut Enumeration Schemes

Cut-set Merging Cut-set merging is the standard bottom-up cut enumeration algorithm. Given a node n and the two cut-sets A and B of its fanin nodes, the cut-merging operation $A \bowtie_k B$ is defined as following (Cong et al., 1999):

$$A \bowtie_k B = \{u \cup v | u \in A, v \in B, |u \cup v| \leq k\} \tag{3.4}$$

Then, all the K-feasible cuts of each node can be enumerated in topological order from PI to PO:

$$\Phi_K(n) = \begin{cases} \{\{n\}\}, & \text{if } n \in \text{PI} \\ \{\{n\} \cup \Phi_K(u) \bowtie_K \Phi_K(v)\}, & \text{otherwise} \end{cases} \tag{3.5}$$

where u and v are the fanins of n, and $\Phi_K(n)$ denote the K-feasible cuts of node n.

Cut Expansion Cut expansion is a top-down cut enumeration scheme (Takata and Matsunaga, 2009). An expansion is to generate a new cut by removing a node in the cut and adding its fanins:

$$EXP(n, \mathcal{C}) = (\mathcal{C} - \{n\}) \cup \{u, v\} \tag{3.6}$$

where u and v are the fanins of n. Any cut but the trivial cut $\{n\}$ can be generated by some expansion. Therefore, all the cuts rooted at n can be enumerated by expanding from $\{n\}$ all the way down to PIs. It is further proved that the expansion range can be compactly narrowed down to the elements in the fanin cuts and this bound is tight (Takata and Matsunaga, 2009).

Related Work on Cut Enumeration

Due to the large number of K-feasible cuts when K is large, some works suggest to heuristically select cuts that are potentially more desired. In area minimization of FPGA mapping (Cong et al., 1999), cuts of a node are accordingly ranked by the mapped area to implement it. Similar idea could be found in a depth-oriented technology mapping algorithm (Mishchenko et al., 2007b), where only a small number (about 4 to 8) of cuts are stored for each node. These *priority cuts* are sorted by different criteria in each mapping pass. Despite of an optimal algorithm (Cong and Ding, 1994) that runs in polynomial time, enumerating only priority cuts enables linear runtime w.r.t. number of nodes, greatly improving its scalability. However, incomplete cut enumeration compromises optimality: although only a small fraction (say 1%) of those cuts is useful, but *a priori* we do not know which 1% (Chatterjee et al., 2006). Therefore, in this paper we focus on exhaustive cut enumeration algorithms.

Parallel cut enumeration has been discussed in a parallel AIG rewriting method (Possan et al., 2018). Although the cut enumeration framework is not fully parallelizable, the authors design

dedicated *cut manager* responsible for providing all necessary routines and data structures for cut computation and storage. Specifically, a node becomes active and can be pushed to the worklist only if its fanin nodes were already processed. Experiments show that parallel cut enumeration scales well with up to 40 threads.

A symbolic cut enumeration method is proposed (Ling et al., 2007) based on binary decision diagram (BDD). The scheme is similar to cut-set merging, but the cuts are represented as BDDs, which enables effective subcut sharing and redundant cut pruning. However, a well-implemented BDD package is required to support the algorithm.

It is worth mentioning that a few simple but useful implementation techniques are introduced (Mishchenko et al., 2007a), which are orthogonal improvements to the cut enumeration algorithms themselves. For example, using signature to test cut properties is usually much faster than directly comparing sets, and the cuts of a node whose fanouts have been processed can be deallocated. These 'lossless' techniques reduce runtime and memory footprint of the algorithm without doing harm to the final result.

3.8.3 Algorithm

Basic Definitions

Before illustrating our algorithm, we first present the definitions we use in the subsequent sections. The core idea is to organize the cut-sets into a tree structure, so that the topological relationship between cuts are preserved. To avoid confusion, we use italic lower case letters $a, b, c, \cdots$ to donote nodes in the original graph, and calligraphic capital letters $\mathcal{A}, \mathcal{B}, \mathcal{C}, \cdots$ to denote nodes and edges in cut-trees (defined in Definition 1), as the latter in fact represent sets.

Definition 1 (Cut-tree)**.** *A cut-tree T_n is a tree where each tree node is associated with a cut of the node n, and each edge stores the (set) difference between its parent node and itself. The cut-tree of a* primary input *(PI) node consists of a single tree node, associated with the* trivial cut*. The cut-tree for a non-PI node is rooted at the trivial cut with a single child, whose sub-tree is the product (see Definition 2) of the cut-trees of its fanin nodes.*

Remark. Every cut-tree T_n is rooted at the trivial cut $\{n\}$. A cut-tree associated with only K-

feasible cuts is called a *K-feasible cut-tree*. The edge (set difference) can be interpreted as 'expand from' since it depicts how to obtain the child node from the parent node.

The above definition of cut-tree is in a recursive manner. In analogy to cut-set merge, we need to formally define the product of two cut-trees:

Definition 2 (Naive Cut-tree Product)**.** *The (naive) product of two cut-trees T_l and T_r is a new cut-tree T_p defined as follows:*

$$V_{T_p} = \{\mathcal{U} \cup \mathcal{V} | \mathcal{U} \in V_{T_l}, \mathcal{V} \in V_{T_r}\}$$
$$E_{T_p} = \{(\mathcal{U}\mathcal{V}, \mathcal{U}'\mathcal{V}', \delta) | (\mathcal{U}, \mathcal{U}', \delta) \in E_{T_l} \text{ or } (\mathcal{V}, \mathcal{V}', \delta) \in E_{T_r}\}$$

where $\mathcal{U}\mathcal{V}$ denotes the node generated from $\mathcal{U} \cup \mathcal{V}$, and an edge $\langle \mathcal{P}, \mathcal{C}, \delta \rangle$ denotes that $\mathcal{P}$ is the parent of $\mathcal{C}$, with the difference δ.

Remark. The definition is abstract. From the implementation perspective, one can traverse the left cut-tree T_l in arbitrary order, and merge ('graft') the right cut-tree to each node of the left cut-tree. By merging I mean taking the union of the cut in the left tree (called *left cut* hereafter) and each cut in the right cut-tree. Note that the product of two K-feasible cut-trees does not necessarily yield a K-feasible cut-tree.

Cut Enumeration through Cut-tree Product

The naive cut-tree product procedure, as introduced in Definition 2, serves as the core scheme for our cut enumeration algorithm. We further propose several pruning strategies during the product of two cut-trees to skip unnecessary computations.

Proposition 1 (Subtree Pruning)**.** When merging a left cut and the right cut-tree, if an edge $\mathcal{E}$ in the right cut-tree contains a node that appears in the left cut, then the whole subtree rooted at the destination (child node) of $\mathcal{E}$ can be pruned.

Proposition 2 (Path Compression)**.** If the associated cut of a cut-tree node $\mathcal{V}$ is not K-feasible, then the node can be pruned, and its children can be directly appended to the parent node of $\mathcal{V}$.

Remark. Recall that an edge stores the difference between the parent node and the child node. Specifically, if we compress a node $\mathcal{V}$ due to its infeasibility, and there were edges $\langle \mathcal{U}, \mathcal{V}, \delta_{uv} \rangle$ and

$\langle \mathcal{V}, \mathcal{W}, \delta_{vw} \rangle$, then there will be a new edge $\langle \mathcal{U}, \mathcal{W}, \delta_{uv} \cup \delta_{vw} \rangle$ after path compression.

Theorem 1. *Cut-tree Product with subtree pruning and path compression enumerates all K-feasible cuts.*

Proof. Correctness of the algorithm can be easily verified if no pruning is presented, as it is equivalent to cut-set merge. We now prove that the algorithm still enumerates all K-feasible cuts with the two pruning strategies.

Without loss of generality, we first suppose the edge $\mathcal{E} = \langle \mathcal{P}, \mathcal{C}, \{d\} \rangle$ that stores a unit set causes pruning. Due to the pruning condition, the left cut $\mathcal{U}$ also contains d. Suppose subtree pruning prunes a right cut $\mathcal{V}$ that should actually be included. Consider the edge from $\mathcal{V}$'s parent to $\mathcal{V}$ that stores $\{p\}$ (i.e., $\mathcal{V}$ can be generated from expanding p), there are two possibilities: p is a successor of d, or p is not a successor of d. In both cases the cut set $\mathcal{U} \cup \mathcal{V}$ is redundant, as there must exist a node $\mathcal{U}_d$ in the left cut-tree that expands d. In the former case, there also exists a node $\mathcal{U}_p$ in the left cut-tree (a successor of $\mathcal{U}_d$), that also expands p, and we have $\mathcal{U}_p \cup \mathcal{V} = \mathcal{U} \cup \mathcal{V}$. An exception would be that $\mathcal{U}_p$ is not K-feasible so that it is pruned, but in such case $\mathcal{U} \cup \mathcal{V}$ cannot be k-feasible as well because $|\mathcal{U} \cup \mathcal{V}| = |\mathcal{U}_p \cup \mathcal{V}| \geq |\mathcal{U}_p| > K$. Similar argument holds for the latter case: if p is not a successor of d, there must exist a node $\mathcal{V}_p$ that does not expand d but expands p. To be specific, $\mathcal{V}_p$ locates in a sibling subtree of $\mathcal{C}$.

We then argue that any cut pruned by path compression is unnecessary for further operation. This is clear due to the monotonicity of set merge: if $\mathcal{V}$ is pruned because $|\mathcal{V}| > K$, then merging $\mathcal{V}$ with any set $\mathcal{U}$ yields an even larger (thus infeasible) set $\mathcal{V} \cup \mathcal{U}$.

It remains to do a bit technical work to verify whether the two pruning strategies can be combined. It turns out that all the key elements are already discussed in the above arguments and we will omit the details due to the limited page length. □

Comparisons with Other Cut Enumeration Schemes

In this section, we compare the efficiency of various cut enumeration schemes. We first define the metric to measure the efficiency of a cut enumeration algorithm, and then prove that cut-tree product is more efficient than cut-set merge and cut expansion.

Definition 3 (Workload of Cut Enumeration)**.** *Workload of a cut enumeration scheme is the total number of cuts (no matter valid or not) computed by the algorithm.*

Remark. Workload is defined as the total number of cuts, rather than counting only feasible ones, because the runtime is roughly proportional to the number of sets computed, even though some of them will be pruned since they are dominated or infeasible.

Theorem 2. *The workload of cut-tree product is less than cut-set merging.*

Proof. The two schemes are exactly equivalent if no pruning is presented: they both compute the merge-product of the cuts of the fanin nodes, the results of which are the same as shown in Theorem 1. Cut-tree enables pruning whenever a reconvergence point exists, reducing its workload by subtree pruning. Therefore the cuts generate by cut-tree product is a subset of cut-set merge, thus the lower workload. □

Theorem 3. *The workload of cut-tree product is less than cut expansion.*

Proof. We prove the theorem by showing that every cut generated by cut-tree product will be generated by cut expansion. In the following two cases the above argument is violated:

1. Cut-tree product generates the same cut twice
2. Cut-tree product generates a cut that cut expansion does not

We first show that the first case does not happen. Consider the path from the root to a node $\mathcal{V}$, the union of all edges along the path shows how can the node $\mathcal{V}$ expanded from the root (as in cut expansion). In fact, this union result for each node is distinct (as will be proved in Lemma 1). Therefore cut-tree product will not generate a cut set twice.

The second case never happens as well. In cut expansion, the nodes are exhaustively expanded until the 'lowest' supports of the fanin cuts are reached. In cut-tree product, the generated cuts contain only the elements in the fanin cuts (expect the node itself), which naturally meets the condition in Lemma 1 of (Takata and Matsunaga, 2009). Given that, all the sets generated by cut-tree product will also be generated by cut expansion.

By the above arguments, we conclude our claim. On top of that, if there are reconvergent points in the original graph, it is likely that cut expansion has to enumerate infeasible cuts to generate feasible cuts, while path compression enables cut-tree product to exclude those cuts without affecting the generation of other cuts, reducing its workload. □

Lemma 1 (Distinct Path Union)**.** *Path Union of a node $\mathcal{V}$ in cut-tree is the union of all the edges along the path from the root to $\mathcal{V}$. In cut-tree product the path union of each node is distinct.*

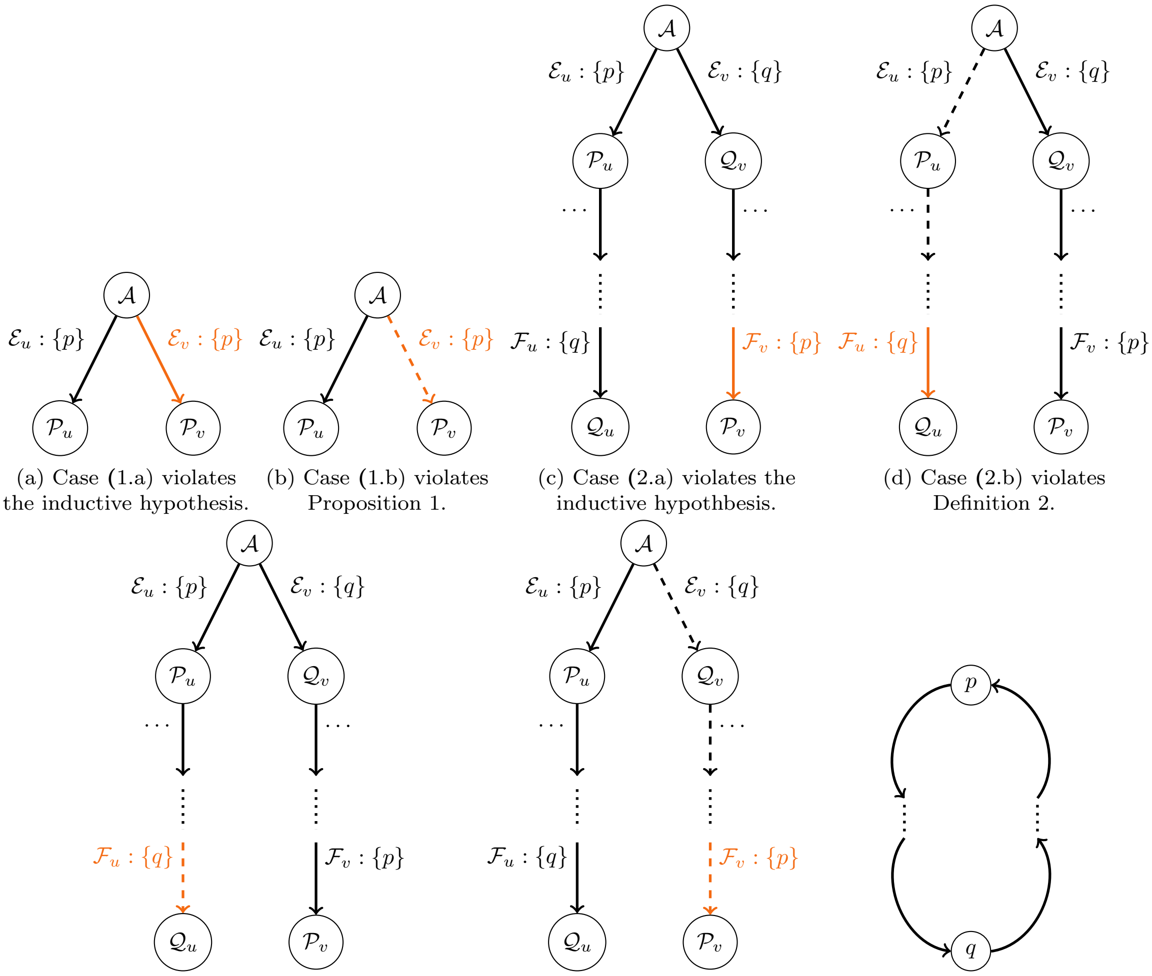
Proof. We prove by induction. The claim trivially holds for the cut-trees of PIs. Suppose the claim also holds for both fanin cut-trees T_l and T_r of a node n. Now suppose for contradiction that there are two distinct nodes $\mathcal{U}$ and $\mathcal{V}$ in the cut-tree T_n, such that the path union of the two nodes are identical. Consider the nearest common ancestor $\mathcal{A}$ of $\mathcal{U}$ and $\mathcal{V}$, i.e., the 'branching' node that makes $\mathcal{U}$ and $\mathcal{V}$ distinct. Consider the two (distinct) edges $\mathcal{E}_u$ and $\mathcal{E}_v$ coming out from $\mathcal{A}$, connecting the path to $\mathcal{U}$ and $\mathcal{V}$ respectively, at least one of the following two cases must happen:

1. $\mathcal{E}_u = \langle \mathcal{A}, \mathcal{P}_u, p \rangle$, $\mathcal{E}_v = \langle \mathcal{A}, \mathcal{P}_v, p \rangle$.
2. $\mathcal{E}_u = \langle \mathcal{A}, \mathcal{P}_u, p \rangle$, $\mathcal{E}_v = \langle \mathcal{A}, \mathcal{Q}_v, q \rangle$, and there exists two other edges $\mathcal{F}_u = \langle \cdot, \cdot, q \rangle$, $\mathcal{F}_v = \langle \cdot, \cdot, p \rangle$, on the path from $\mathcal{C}$ to $\mathcal{U}$ and from $\mathcal{C}'$ to $\mathcal{V}$, respectively.

Otherwise, the union results cannot be identical. Let $\mathcal{U}$ be generated from merging $\mathcal{U}_l$ and $\mathcal{U}_R$, $\mathcal{V}$ generated from merging $\mathcal{V}_l$ and $\mathcal{V}_r$. Let $\mathcal{R}_l$ and $\mathcal{R}_r$ be the root of T_l and T_r, respectively. According to Definition 2, the edges in T_n must be inherited from T_l or T_r. All the cases discussed below are shown in Figure 3.7.

The first case should not happen. (**1.a**) If $\mathcal{E}_u$ and $\mathcal{E}_v$ are inherited from the same cut-tree (no matter T_l or T_r), then the inductive hypothesis on the fanin cut-trees is violated. (**1.b**) On the contrary, if they are inherited from different trees, say $\mathcal{E}_u$ from T_l and $\mathcal{E}_v$ from T_r, since p is an element in $\mathcal{A}$, $\mathcal{E}_v$ should be pruned according to the rule of subtree pruning (Proposition 1).

The second case never happens as well. Depending on where are the 4 edges inherited from, there are 16 (2^4) cases. (**2.a**) Similar to the above argument, the four edges could not be inherited from the same cut-tree, otherwise the inductive hypothesis on the fanin cut-trees is already violated. (**2.b**) According the cut-tree product order (Definition 2), it is impossible that $\mathcal{E}_u$ ($\mathcal{E}_v$) is inherited



(a) Case (1.a) violates the inductive hypothesis.
(b) Case (1.b) violates Proposition 1.
(c) Case (2.a) violates the inductive hypothbesis.
(d) Case (2.b) violates Definition 2.
(e) Case (2.c) violates Proposition 1.
(f) Case (2.d) violates Proposition 1 or the definition of DAG.
(g) The original graph in case (2.d) is cyclic.

Figure 3.7: All counterexamples that in fact never happen. A solid arrow denotes an edge inherited from the left fanin cut-tree T_l, while a dashed edge is inherited from the right fanin cut-tree T_r. An edge in orange violates the inductive hypothesis or some definition/proposition.

from T_r and $\mathcal{F}_u$ ($\mathcal{F}_v$) from T_l. (**2.c**) Noticing that $p, q \in \mathcal{A}$, if $\mathcal{E}_u$ ($\mathcal{E}_v$) is inherited from T_L and $\mathcal{F}_u$ ($\mathcal{F}_v$) from T_R, the subtree pruning rule (Proposition 1) is violated. (**2.d**) If $\mathcal{E}_u$ and $\mathcal{F}_u$ are inherited from T_L (T_R) and $\mathcal{E}_V$ and $\mathcal{F}_v$ are from T_R (T_L), either $p, q \in \mathcal{A}_L$ and $p, q \in \mathcal{A}_\mathcal{R}$ (the same as **2.c**), or q is a TFI of p and p is a TFI of q, which cannot be true in an acyclic graph. The above cases **2.a** - **2.d** cover all the possibilities, and thus we conclude our claim.

□

3.9 Summary

Identifying arithmetic blocks is a vital procedure for various tasks like malicious logic detection and logic optimization. In this chapter, we propose a graph learning-based arithmetic block identification framework that efficiently recognizes the boundary of arithmetic blocks. To boost the performance of the whole framework, we propose a specialized graph neural network architecture for DAG representation learning, which outperforms existing dominantly used GNNs. We further come up with a network flow approach to match input and output wires predicted by the GNN model. Experimental results have confirmed the excellent performance of our framework: compared with state-of-the-art functional, structural and machine learning-based block mapping schemes, our framework achieves the highest sensitivity with the fastest runtime in adder identification from an open-source RISC-V CPU design (the Rocket core). We also carried out a comprehensive ablation study to analyze the effectiveness of the proposed techniques. Finally, we discussed a new cut enumeration scheme based on cut-tree product.

Chapter 4

Reinforcement Learning Driven Floorplan Optimization

4.1 Motivation

Electronic Design Automation (EDA) lies at the heart of modern computer science technologies. Various computationally challenging (viz. NP-hard) problems gave birth to the exciting progress in solving techniques, most among which are hand-crafted heuristics that are carefully designed by domain experts and scientists. Yet, the fantasy of automatic algorithm design for difficult problems has never been shattered.

Reinforcement learning has recently offered a promising direction for such a dream. As introduced in Section 2.2, typical floorplanning algorithms make use of various data structures to represent the geometric relation between modules, and construct or perturb such data structures to obtain good floorplan results. Despite of that, specialized knowledge is a must for a successful design, as well as a considerable amount of trial-and-errors, prohibiting the development of new algorithms in some sense. Besides, as the transistor technology node scaling down, modern circuit design has become much more complicated, and the ever-increasing number of modules in a chip brings about the scalability issue, emphasizing the demand for effective algorithms that work well on large-scale

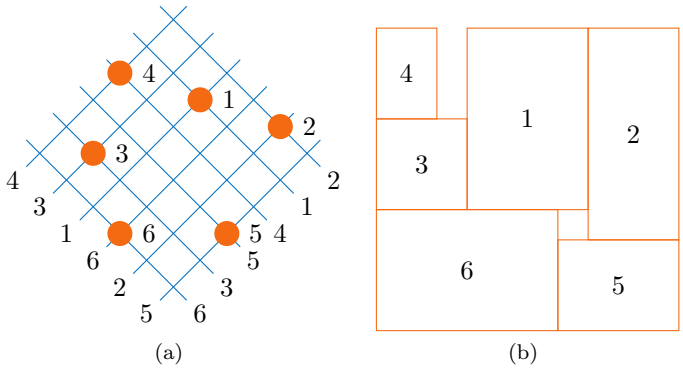


Figure 4.1: (a) The oblique grid (Tang and Wong, 2001) shows the relative position between blocks for sequence pair (<4 3 1 6 2 5>, <6 3 5 4 1 2>); (b) The corresponding packing. The dimensions for the 6 blocks are: 1(4×6), 2(3×7), 3(3×3), 4(2×3), 5(4×3), 6(6×4).

cases. All the above problems motivate us to utilize reinforcement learning for automatic algorithm design. Unlike previous works that adopt an existing local search algorithm and improve a few settings of that algorithm with RL, our work takes one step forward: we aim to acquire a local search algorithm from the scratch, i.e., we avoid to introduce too much prior human knowledge during the search that might mislead the learning.

4.2 Preliminaries

4.2.1 Floorplan

In general, floorplanning is to generate relative locations for modules. Given a set of n rectangular blocks $B = \{b_1, b_2, \cdots, b_n\}$ and a netlist N specifying their connections, a floorplan F seeks a planar location assignment $(\boldsymbol{x}, \boldsymbol{y})$ of B, providing no module overlap, to minimize the total chip area and to reduce the total wirelength.

Based on area function $A(\cdot)$ and wirelength function $W(\cdot)$, the optimization problem is formulated as follows

$$\begin{aligned} \min_{F} \quad & A(F) + \alpha W(F) \\ \text{s.t.} \quad & F \text{ is a legal solution.} \end{aligned} \tag{4.1}$$

Definition 4 (Sequence Pair Representation)**.** *A sequence pair* (Γ_+, Γ_-) *is a pair of sequences of* n *elements that imposes the relationship between each pair of blocks as follows (Tang and Wong,*

2001):

$$(< ..b_i..b_j.. >, < ..b_i..b_j.. >) \implies b_i \textit{ is to the left of } b_j;$$

$$(< ..b_j..b_i.. >, < ..b_i..b_j.. >) \implies b_i \textit{ is below } b_j.$$

As an example, Figure 4.1 shows the imposed relationship by the sequence pair (<4 3 1 6 2 5>,<6 3 5 4 1 2>) in an oblique grid. It can be inferred from the figure that all the imposed relationship constraints are satisfiable, see (Murata et al., 2003) for the prove.

In fact, given any sequence pair, one of the area-optimal packing subject to the constraints can be obtained in $\mathcal{O}(n \log \log n)$ time (Tang and Wong, 2001) based on a fast longest common subsequence computation.

4.2.2 Local Search

Local search is a popular heuristic method for solving combinatorial optimization problems. Roughly speaking, a local search algorithm starts off with an initial solution and then continually tries to find better solutions by searching neighbourhoods (Aarts et al., 2003).

Simulated annealing (SA) (Kirkpatrick et al., 1983) is a probabilistic technique inspired from annealing in metallurgy. Formally, let S be the finite set of all complete solutions, $\mathcal{E} : S \to \mathbb{R}$ be an energy function defined on S, and $\mathcal{N} : S \to S$ be a neighbor function. Note that for each $s \in S$,$\mathcal{N}(s) \subset S - \{s\}$. SA starts at a state $s \in S$. At each step, SA considers some neighboring state $s' \in \mathcal{N}(s)$ of the current state s, and decides between moving the system to state s' or staying in state s based on the acceptance probability given as follows:

$$P(s', s, T) = \exp\,[\frac{1}{T} \max\,(0, \mathcal{E}(s') - \mathcal{E}(s))], \tag{4.2}$$

where $T \in \mathbb{R}$ is the temperature to control how 'bad' moves are accepted. These probabilities ultimately lead the system to move to states of lower energy.

4.2.3 Basis of Reinforcement Learning

Reinforcement learning learns the mapping from states to actions, so as to maximize a numerical reward signal (Sutton and Barto, 2018). We use the agent-environment interface (Sutton and Barto, 2018) to describe it specifically: at each time step t, the agent receives state s_t that represents the

current state of the environment, and on that basis selects an action a_t. One time step later, the agent receives a numerical reward $R_{t+1} \in \mathbb{R}$, and finds itself in a new state s_{t+1}. This framework is a considerable abstraction of the goal-directed learning problem from interaction.

S is a set of states to describe the environment;

A is a set of actions that the agent can take;

P is the transition probability between states under actions;

R is the immediate reward signal.

MDP is a discrete-time stochastic control process: at timestamp t, the agent is at a state s_t; then the agent selects an action $a_t \in A$, and finds itself in a new state s_{t+1}, with an immediate reward r_{t+1}. By concatenating states, actions, and rewards of all time steps together, we obtain a sequence called an episode: $s_0, a_0, r_1, s_1, a_1, \cdots$, which fully describes the trajectory of the agent.

The objective in MDP is to find a good policy $\pi(s)$ to maximize the accumulative reward (return)

$$G_t = \sum_{t}^{\infty} \gamma^t r_t, \tag{4.3}$$

where $\gamma \in [0, 1]$ is the discount factor to penalize future rewards. We define the value of a state s under policy π (denoted as $V_\pi(s)$) as the expected return starting from that state. According to *Bellman Equation*, the state value can be stated recursively:

$$\begin{aligned} V_\pi(s) &= \mathbb{E}[G_t | S_t = s] \\ &= R(s, \pi(s)) + \gamma \sum_{s'} P(s, \pi(s), s') V_\pi(s'). \end{aligned} \tag{4.4}$$

Similarly, we define the expected return from taking an action a in state s under policy π as the action value:

$$\begin{aligned} Q_\pi(s, a) &= \mathbb{E}[G_t | S_t = s, a_t = a] \\ &= R(s, a) + \gamma \sum_{s'} P(s, a, s') V_\pi(s') \\ &= R(s, a) + \gamma \sum_{s'} P(s, a, s') \mathbb{E}_{a' \sim \pi} Q_\pi(s', a'). \end{aligned} \tag{4.5}$$

4.2.4 Common Reinforcement Learning Approaches

Dynamic Programming

If the model is fully known (i.e., P and R of the MDP is known), and both the state space and action space are finite, solutions to the MDP can be obtained through iterative policy improvement following bellman equation. The *generalized policy iteration* (GPI) algorithm refers to the optimization process consisting of iterative policy evaluation (through Equation (4.4)) and policy improvement by greedily selecting the action with highest action value:

$$\pi'(s) = \underset{a}{\operatorname{argmax}} \left\{ R(s,a) + \gamma \sum_{s'} P(s,a,s') V_\pi(s') \right\}. \tag{4.6}$$

Through alternating policy evaluation and policy improvement,

$$\pi_0 \xrightarrow{\text{evaluate}} V_{\pi_0} \xrightarrow{\text{improve}} \pi_1 \xrightarrow{\text{evaluate}} \cdots \xrightarrow{\text{improve}} \pi^* \xrightarrow{\text{evaluate}} V^*,$$

the policy is guaranteed to be better:

$$\begin{aligned} & Q_\pi(s, \pi'(s)) \\ =& Q_\pi(s, \underset{a}{\operatorname{argmax}} \left\{ R(s,a) + \gamma \sum_{s'} P(s,a,s') V_\pi(s') \right\}) \\ =& \max_a Q_\pi(s,a) \geq Q_\pi(s, \pi(s)). \end{aligned} \tag{4.7}$$

Monte Carlo Methods

Monte Carlo methods rely on large amount of repeated random sampling to obtain statistical properties, which are broadly used in various optimization and simulation problems. Following the GPI framework, policy evaluation can be substituted by Monte Carlo simulations, while policy improvement remains the greedy behaviour.

Recall from Equation (4.4) that $V_\pi(s) = \mathbb{E}[G_t | S_t = s]$, we can empirically estimate G_t through sampling instead of exactly computing the expectation:

$$v_\pi(s) = \frac{\sum_{t=1}^{T} \mathbb{1}[s_t = s] G_t}{\sum_{t=1}^{T} \mathbb{1}[s_t = s]}, \tag{4.8}$$

where $\mathbb{1}[\cdot]$ is the indicator function. Similarly we can estimate $Q_\pi(s, a)$ as well:

$$q_\pi(s, a) = \frac{\sum_{t=1}^{T} \mathbb{1}[s_t = s, a_t = 1] G_t}{\sum_{t=1}^{T} \mathbb{1}[s_t = s, a_t = a]}. \tag{4.9}$$

With sufficiently amount of samples, the procedure is able to precisely estimate the values due to the law of large numbers.

In Monte Carlo methods, state values and action values are estimated through raw experience without knowing the model dynamics, which is often the case in reality. However, it is important to note that Monte Carlo methods only work in episodic problems (finite episode length), otherwise the return may not be correctly computed. In practice, the method is often regarded as sample inefficient, in that it requires complete episodes to update estimates.

Temporal Difference Learning

Similar to Monte Carlo methods, temporal difference (TD) Learning is another model-free learning algorithm that learns from raw experiences. The key difference between the two methods is that TD is based on bootstrapping: the values are estimated with regard to other estimates, rather than exclusively relying on actual rewards.

From bellman equation, we know that $R(s_t, \pi(s_t)) + V_\pi(s_{t+1})$ is an unbiased estimation of state value $V_\pi(s_t)$. This motivates the following update rule:

$$v_\pi(s_t) = v_\pi(s_t) + \alpha(R(s_t, \pi(s_t)) + \gamma v_\pi(s_{t+1}) - v_\pi(s_t)), \tag{4.10}$$

where α is the step size (a.k.a. learning rate), and $R(s_t, \pi(s_t)) + \gamma v_\pi(s_{t+1})$ is refereed to as the TD target. Similarly, we can estimate $Q_\pi(s_t, a_t)$ as well:

$$\begin{aligned} q_\pi(s_t, a_t) = q_\pi(s_t, a_t) \\ + \alpha(R(s_t, a_t) + \gamma q_\pi(s_{t+1}, a_{t+1}) - q_\pi(s_t, a_t)). \end{aligned} \tag{4.11}$$

The algorithm that uses Equation (4.11) to estimate action value in GPI is known as SARSA in the literature.

A popular algorithm, Q-learning (Watkins and Dayan, 1992), is an off-policy variant of TD learning. In an off-policy algorithm, the policy that the agent follows to interact with the environment

Table 4.1: Comparisons of value-based reinforcement learning algorithms. Methods differ in whether it samples and whether it boostraps.

Method	Sampling?	Bootstrapping?
Dynamic Programming	No	Yes
Monte Carlo Method	Yes	No
Temporal Difference Learning	Yes	Yes
Brute-force	No	No

(behavior policy) is independent of the optimal policy that the agent aims to learn (target policy). To see the difference, in SARSA (or Equation (4.11)), a_{t+1} is selected following the target policy π, which the algorithm is going to estimate. In Q-learning, however, the action is greedily selected, despite the fact that the target policy π is actually not greedy:

$$\begin{aligned} q_\pi(s_t, a_t) = q_\pi(s_t, a_t) \\ + \alpha(R(s_t, a_t) + \gamma \max_a q_\pi(s_{t+1}, a) - q_\pi(s_t, a_t)). \end{aligned} \tag{4.12}$$

It is interesting to note that taking max over all actions may overestimate the action values. Please refer to (Sutton and Barto, 2018) for more discussions.

So far, we have introduced three value-based reinforcement learning approaches. As shown in Table 4.1, we compare the three algorithms, together with brute-force, in terms of whether it samples and whether it boostraps.

Policy Gradient

Instead of estimating state or action values and deciding policy according to the values, an alternative approach is to directly search in the policy space. Suppose the policy is parameterized by θ, the objective is still to maximize the accumulative reward:

$$J(\theta) = \mathbb{E}[V_\theta(s_0)]. \tag{4.13}$$

As by Policy Gradient Theorem (Sutton et al., 2000), we have:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[Q_\pi(s, a) \nabla \ln \pi(a|s; \theta)]. \tag{4.14}$$

The algorithm REINFORCE (Williams, 1992) can be regarded as the Monte Carlo policy gradi-

ent. Recalling that $Q_\pi(s_t, a_t) = \mathbb{E}[G_t|s_t, a_t]$, REINFORCE estimates $\nabla_\theta J(\theta)$ by $G_t \nabla \ln \pi(a_t|s_t;\theta)$ and updates the policy parameters by

$$\theta = \theta + \alpha\gamma^t G_t \nabla \ln \pi(a_t|s_t;\theta), \tag{4.15}$$

for each $t = 1, \cdots, T$ in the episode.

Actor Critic

In vanilla policy gradient algorithms, the variance of the gradient could be large. To stabilize the training, one idea is to utilize the value model to guide the policy search, namely the actor-critic (Konda and Tsitsiklis, 2000) framework. In short, the critic is trained to predict the state or action values using TD learning, and the actor is trained to select actions by policy gradient using the predicted values.

4.2.5 State-of-the-art Reinforcement Learning Algorithms

A great many variants of the above basic reinforcement learning approaches have been proposed in recent years. Common techniques include experience replay (Lin, 1992) and prioritized experience replay (Schaul et al., 2016) to both break sample correlations and to improve sample efficiency, multi-step learning (Sutton, 1988) to accelerate training, trust region optimization (Schulman et al., 2015) to constrain step direction, and maximum entropy reinforcement learning (Ziebart, 2010) to encourage exploration.

There is no silver bullet in reinforcement learning algorithm selection. However, the first thing to consider is often in the action space. For discrete action space, one may consider deep Q-learning (DQN) (Mnih et al., 2013) and its variants(e.g. rainbow (Hessel et al., 2018) that combines lots of useful techniques), or advantage actor-critic (A2C) (Mnih et al., 2016) and its variants (e.g. actor-critic with experience replay (ACER) (Wang et al., 2017)). For continuous action space, one popular choice is soft actor-critic (SAC) (Haarnoja et al., 2018), and other alternatives include proximal policy optimization (PPO) (Schulman et al., 2017) and Twin-delayed DDPG (TD3) (Fujimoto et al., 2018). Among the above, A2C and PPO are on-policy algorithms while the rests are off-policy. In summary, lots of the newly proposed algorithms aim to reduce variance to stabilize training.

4.3 Algorithm

The purpose of this work is to acquire an effective local search heuristic, with which good floorplans could be obtained through local search. In this section, we will first formulate the above problem as a reinforcement learning problem, and then we discuss how to select features to represent the states, and how to construct an agent that makes decision.

4.3.1 Local Search as a Reinforcement Learning Problem

We first formally define the MDP for our local search problem.

State Space As introduced in Section 4.2.2, a state s is a complete solution. Here a complete solution includes both sequences Γ_+, Γ_-, as well as the orientation of each block.

Action Space Several perturbations are defined on a solution s to generate $\mathcal{N}(s)$. At each step, a subset of neighbours $s' \subset \mathcal{N}(s)$ are sampled, and the agent decides to accept one of the neighbours or reject to move. Therefore, the action space size is the number of sampled neighbours plus one (for the reject). We allow 6 types of perturbation on a solution:

1. Exchange two blocks in Γ_+.
2. Exchange two blocks in Γ_-.
3. Exchange two blocks in both Γ_+ and Γ_-.
4. Delete one block and insert back to a random position in both Γ_+ and Γ_-.
5. Rotate one block by $90°$.
6. Flip one block.

Transition The transition model of our MDP is deterministic, viz., no randomness is involved once the state and the action are given. For this reason, we reload the transition model with signature $\mathcal{T} : S \times A \to S$. Hereafter, we use $s' \leftarrow \mathcal{T}(s, a)$ to denote a state transition.

Reward Assigning rewards to the actions is critical in reinforcement learning. Since the agent seeks to maximize the total reward, the action of maximizing total rewards should be consistent with the target of the problem. Basically, the rewards are assigned whenever a global better solution is found. We use the reduction of energy (i.e., $\Delta\mathcal{E}$) as the reward, so that maximizing the total reward $\sum \Delta\mathcal{E}$ is equivalent to minimizing the cost. The reward is normalized to $[0, 1]$ by comparing with the energy of the initial state:

$$R = \frac{\Delta\mathcal{E}}{baseline}. \tag{4.16}$$

Intuitively, the reward encourages the design with lower cost, which aligns with our target of reducing area and wirelength.

To accelerate training, we empirically add two adversarial rewards to discourage useless explorations. First, if the agent decides to reject, while one of the sampled neighbour has lower energy than the current state, a negative reward of -0.01 is given. In other words, the agent is always encouraged to move to a neighbor state with lower energy. Second, if the agent accepts a state, whose energy is higher than 1.2 times of the lowest energy neighbor, a negative reward of -0.01 is given. This is to discourage the agent to search a high energy region that can hardly contain a good solution.

4.3.2 Features

As mentioned above, neighbour solutions are sampled from all the 6 types of perturbation. To help the agent make better decisions, providing helpful features to guide the local search is critical. Here three sets of features are included:

- **Energy**: The central goal of the local search is to find a state with minimal energy, which is defined as the negative cost given by Equation (4.1) after packing all the blocks. In simulated annealing, accepting or rejecting a move is based on current state energy $\mathcal{E}(s)$ and the neighbor state energy $\mathcal{E}(s')$. Besides that, we provide a set of other energy-related statistical information, including the lowest and average energy through the search path ($\mathcal{E}(s^*)$ and $\overline{\mathcal{E}}$), the average energy on the search path since the lowest energy state ($\overline{\mathcal{E}^*}$), and the lowest energy among the sampled neighbors ($\mathcal{E}'_*$).

Table 4.2: Features for decision making. All the items are normalized to $[-1, 1]$ or $[0, 1]$.

Index	Item	Range	Description
0	$\mathcal{E}(s)$	[-1, 1]	Current energy
1	$\mathcal{E}(s')$	[-1, 1]	Neighbor energy
2	$\mathcal{E}(s^*)$	[-1, 1]	Lowest energy so far
3	$\overline{\mathcal{E}}$	[-1, 1]	Average energy
4	$\overline{\mathcal{E}^*}$	[-1, 1]	Average energy since s^*
5	$\mathcal{E}'_*$	[-1, 1]	Lowest energy of sampled neighbours
6	$Area(b_i)$	[0, 1]	Size of the perturbed block
7	aff	[0, 1]	Number of effected blocks
8	t	[0, 1]	Search progress

- **Effect**: How does a move affect the whole floorplan solution? We identify the effect by both the size of the perturbed block(s), as well as the number of blocks that are affected. A block is considered affected if and only if the location of the block is related to the perturbed block(s). For example, if we rotate one block, those blocks above or on the right of this block will be affected. Other situations are also analyzed according to the packing of sequence pair.

- **Progress**: Intuitively, the search progress has a similar role to the temperature scheduling. Therefore we included the search progress as a feature. Although we also included an early-stop mechanism, the progress of early-stopping is not visible to the agent because it is considered irrelevant to the local search.

We list in Table 4.2 the detailed features, which we concatenate into a vector to feed the agent. Note that all the features are normalized for easier training. For this purpose, we denote the energy of the initial state as e_0, and normalize all the energy related entries by $\min(1, \mathcal{E}/e_0 - 1)$.

In practice, we realize that including noisy features does harm to, or at least slows down training. For example, we tried to include an indicator to tag the perturbation type, which turns out to be a noise as an input feature.

4.3.3 Neural Network as the Agent

Recall that the policy π of an agent is a mapping from the state space to the action space. A straightforward way to construct such a policy is to store the best action of each possible state. However, a quick estimation shows that for n blocks, there are totally $(n!)^2 \times 8^n$ different states (two

permutation and rotation/flip of the blocks), which makes the tabular method infeasible. Therefore, we utilize a neural network as the policy approximator. Since the input features are concatenated into an 1D vector, a simple multi-layer perceptron (MLP) should be good in our case, where the input dimension equals to the feature dimension, and the output dimension is always 1 for the value prediction. A ReLU layer is inserted after all but the last layers as the activation. The neural network is trained with back-propagation, as will be illustrated in detail in Section 4.4.

4.4 Training

4.4.1 Dealing with Large Action Spaces

Reasoning in an environment with a large number of possible discrete actions is always challenging, as exploring a large action space will be unstable and inefficient, and thus requires much more training efforts. As concrete examples, there will be totally 16575 possible actions in each state for a floorplan problem on 25 blocks, and 1015050 actions for 100 blocks. Prior work proposed several strategies to improve learning in large action spaces. Factorizing the action space into binary subspaces (Pazis and Parr, 2011) is natural for the cases with many action variables or with a finely discretized continuous action space. However the method requires a fixed size action space, while our action space size is a polynomial of the size of the problem. Other paper suggested embedding the discrete actions into a continuous space (Dulac-Arnold et al., 2015) and eliminating actions with an extra training signal (Zahavy et al., 2018), both of which aim to prune irrelevant solutions to improve the speed and the quality of training. Despite of that, evaluating an action is extremely costly in our local search formulation, since both the energy of a state and the estimated value of the action need to be calculated, making the above solutions still intractable. Inevitably, we have to sample actions during both training and testing.

4.4.2 Deep Q-Learning

We use deep Q-learning (Mnih et al., 2013) (DQN) to estimate the value of each move. Following the common practice, we define the value q of taking action a at state s under policy π as the expected

return starting from that state:

$$q_\pi(s,a) := \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \big| S_t = s, A_t = a \right], \tag{4.17}$$

where t is the timestamp and γ is the discounting factor. The optimal action value is therefore given by $q^*(s,a) := \max_\pi q(s,a)$. As discussed in 4.3.3, we use a neural network to approximate the action value, $Q(s,a;\theta) \approx q^*(s,a)$. The loss function is defined according to the *bellman equation*:

$$\begin{aligned} L(\theta_i) &= \mathbb{E}_{s,a\sim\rho(\cdot)} \left[(y_i - Q(s,a;\theta_i))^2 \right] \\ &= \mathbb{E}_{(\cdot)} \left[\Big(r + \gamma \max_{a'} Q(s',a';\theta_{i-1}) - Q(s,a;\theta_i) \Big)^2 \right], \end{aligned} \tag{4.18}$$

where $\rho(\cdot)$ is the behaviour distribution over s and a, and y_i is called the target of the update. Then we can update the model through back-propogating the loss with respect to the weights ($\nabla_{\theta_i} L(\theta_i)$). DQN is considered as *off-policy* because it estimates the greedy policy ($\max_a(Q(s,a;\theta))$, while samples the state space following a behaviour policy. The overall algorithm is listed as Algorithm 1.

We select to use an off-policy algorithm for the following reasons:

1. A local search episode is typically very long (e.g. 10k steps). The consecutive moves are highly correlated, which does harm to training. Instead, in an off-policy algorithm we sample experiences from the replay memory to break the strong correlations and thus reduce the variance.

2. Sampling is not free. Packing and calculating wirelength is somehow time-consuming compared to running a model. With the replay memory, each data sample is potentially used many times in training, which increase data efficiency.

3. Exploration is necessary. If we learn online a greedy policy that picks the action with the largest expected return, then we always select the same action during training. An off-policy algorithm naturally decouples sampling behaviour and the learnt policy, allowing more random exploration in training while still being greedy in testing.

Algorithm 1 DQN for Floorplan Local Search

function TRAIN()
  Initialize replay memory D;
  Initialize Q with random weights;
  Initialize ϵ;
  for episode $\leftarrow 1, \ldots, M$ **do**
    Reset $s \leftarrow s_1$;
    for $t \leftarrow 1, \ldots, T$ **do**
      if $\epsilon > u \sim \mathcal{U}(0, 1)$ **then**
        Sample a random action a_t;
      else
        Select $a_t \leftarrow \max_a Q(s_t, a_t; \theta)$;
      end if
      $s_{t+1} \leftarrow \mathcal{T}(a_t, s_t)$;
      Save experience $(s_t, a_t, r_{t+1}, s_{t+1})$ to D;
      UPDATEMODEL();
      Update ϵ;
    end for
  end for
end function

procedure UPDATEMODEL()
  Sample a batch of $(s_j, a_j, r_{j+1}, s_{j+1})$ from D;
  if s_{j+1} is a terminal state **then**
    $y_j \leftarrow r_{j+1}$;
  else
    $y_j \leftarrow r_{j+1} + \gamma \max_{a'} Q(s_{j+1}, a'; \theta)$;
  end if
  Calculate $L \leftarrow (y_j - Q(s_j, a_j; \theta))^2$;
  Update θ with back-propagation;
end procedure

4.4.3 Convergence Analysis

With the MDP defined (S, A, T, R) in Section 4.3, the Q-learning algorithm is given by the iteration rule

$$\begin{aligned} Q_{t+1}(s_t, a_t) \longleftarrow\ & Q_t(s_t, a_t) \\ & + \alpha(s_t, a_t)\left[r_t + \gamma \max_{b \in A_t} Q_t(s_{t+1}, b) - Q_t(s_t, a_t)\right], \end{aligned} \tag{4.19}$$

where A_t is a subset of the original action space A after random sampling, $\gamma < 1$. Considering that the action space A is extremely large and intractable for efficient training, intuitively we randomly sample the action space in each iteration.

Obviously, $\max_{b \in A_t} Q_t(s_{t+1}, b) \leq \max_{b \in A} Q_t(s_{t+1}, b)$, and thus iteration rule (4.19) will not return us the same Q-function if the action space differs, even if it converges. We will prove that the random iterative process (4.19) converges to a Q-function which is different from the optimal one $Q*$.

Theorem 4. *Given the MDP defined (S, A, T, R) in Section 4.3, the iteration rule* (4.19) *converges with probability 1, as long as constraints*

$$\sum_t \alpha_t(s, a) = \infty \qquad \sum_t \alpha_t^2(s, a) < \infty \tag{4.20}$$

are satisfied for all $(s, a) \in S \times A$.

To establish this theorem some mathematical results from stochastic approximation are required. The following theorem is much more powerful and general than Theorem 4. We present it as a lemma and the proof can be found in (Jaakkola et al., 1994).

Theorem 5. *The random process $\Delta_{t+1}(x) = (1 - \alpha_t(x))\Delta_t(x) + \beta_t(x)F_t(x)$ converges to zero with probability 1 (w.p.1) under the following assumptions:*

1. *The state space is finite.*

2. *The following constraints are satisfied.*

$$\begin{aligned} \sum_t \alpha_t(x) = \infty \qquad & \sum_t \alpha_t^2(x) < \infty \\ \sum_t \beta_t(x) = \infty \qquad & \sum_t \beta_t^2(x) < \infty \end{aligned} \tag{4.21}$$

and $\mathbb{E}[\beta_t(x)|P_t] \leq \mathbb{E}[\alpha_t(x)|P_t]$ *uniformly w.p.1.*

3. $\|\mathbb{E}[F_t(x)|P_t]\|_W < \gamma \|\Delta_t\|_W$, *where* $\gamma \in (0, 1)$.

4. $\mathbf{Var}[F_t(x)|P_t] \leq C(1 + \|\Delta_t\|_W)^2$, *where* C *is a constant.*

Here $P_t = \{\Delta_t, \Delta_{t-1}, \cdots, F_{t-1}, \cdots, \alpha_{t-1}, \cdots, \beta_{t-1}, \cdots\}$ *stands for the past at step* t. $F_t(x)$, $\alpha_t(x)$ *and* $\beta(x)$ *are allowed to depend on the past insofar as the above conditions remain valid. The notation* $\|\cdot\|_W$ *refers to some weighted maximum norm.*

The full proof of Theorem 5 is complicated and out of the scope of this thesis. With the help of Theorem 5, we are able to prove Theorem 4.

4.5 Experimental Results and Discussions

4.5.1 Setup

We implemented the proposed solution in python, while trained the neural model with PyTorch (Paszke et al., 2017). We use Adam (Kingma and Ba, 2015) as the optimizer with an initial learning rate of $5 * 10^{-4}$.

To encourage exploration, the exploration factor ϵ is initially set to be 1. Then it is linearly annealed to 0.1 in the first 15000 steps and fixed afterwards. We use a replay memory of size 20000 that stores the most recent experiences. During training, a batch of 128 experiences are sampled from the replay memory. The discount factor is set to be 0.995. To stabilize training, we fix the target network and synchronize the trained policy network to it every 10 episodes.

4.5.2 Data Generation

Random netlists are generated as training samples. Due to the great difference in problem sizes, we train two models, one of which (the lite one) is for MCNC and the other (the large one) is for GSRC benchmarks. Each netlist for training the lite model consists of 50 blocks with integer width and height in the range of $[10, 100]$. Each block has 3 pins at random locations, and 50 signals are randomly generated, each of which connects 3 random pins. The large network is trained with netist consisting of 250 blocks, 10 pins for each block, and 250 signals. The rest settings are the same.

Table 4.3: Area Minimization on MCNC Benchmark. Our results are directly from minimizing area and wirelength together, while the two other columns are area minimization only. Better results are emphasized in bold.

Circuit	# modules	Area ($\times 10^5$)		
		Ours	SA	FAST-SP (Tang and Wong, 2001)
apte	9	**47.08**	**47.08**	*46.92*
xerox	10	20.42	**20.31**	*19.80*
hp	11	**9.21**	9.26	*8.95*
ami33	33	1.24	**1.22**	*1.21*
ami49	49	38.65	**38.10**	*36.50*

4.5.3 Baseline Tuning

Parameters of simulated annealing greatly affect the performance of the algorithm. For example, a low initial temperature may have the search stuck at a poor local minima, while an over-high temperature just accepts all the moves, walking in the search space in vain. Therefore, we first ran a batch of experiments to tune the parameters of simulated annealing, as listed in Table 4.4. Best obtained result for each case appears in different settings, while in general higher initial temperature and higher number of inner loops yield better results, which is intuitively very reasonable. To this regard, we will use setting 8 in subsequent experiments, where the initial temperature is set to be 10^7, end temperature to be 10^{-10}, inner loop of 200, and an exponential cooling factor of 0.97.

4.5.4 MCNC Benchmark

We tested our agent on the MCNC netlist (Yang, 1991) benchmark.

Area Minimization

We first investigate the problem of area minimization. Since we trained our model on area and wirelength minimization, we directly use the area of the resulted solution to compare. We compare the results of our agent and Simulated Annealing (SA). The results are listed in Table 4.3.

Table 4.4: Tuning Simulated Annealing parameters, including initial temperature, end temperature, number of inner loop, and cooling schedule. Best result (in bold) for each case is obtained in different settings.

Setting	Parameters				Area (mm^2)				
	Init T	End T	Inner loop	Cooling	apte	xerox	hp	ami33	ami49
1	10^6	10^{-10}	50	0.97	48.28	21.77	9.72	1.24	38.87
2	10^7	10^{-10}	25	0.97	**47.08**	20.54	9.38	1.31	39.65
3	10^7	10^{-10}	50	0.95	47.56	20.40	9.63	1.30	39.29
4	10^7	10^{-9}	50	0.97	47.56	20.36	9.33	1.26	40.84
5	10^7	10^{-10}	50	0.97	47.56	20.36	9.33	1.25	40.84
6	10^7	10^{-11}	50	0.97	47.56	20.36	9.33	1.25	40.70
7	10^7	10^{-10}	100	0.97	48.71	21.10	9.21	**1.20**	38.38
8	10^7	10^{-10}	200	0.97	**47.08**	**20.31**	9.26	1.22	**38.10**
9	10^8	10^{-10}	50	0.97	47.52	21.23	**9.17**	1.28	38.72

Area and Wirelength Minimization

Then we switch to the problem of minimizing both area and wirelength. The multi-objectives actually make the problem much more difficult. In area minimization, it is often the case that we switch two internal blocks and the area does not change at all. With wirelength minimization, however, a bad swap may greatly increase the wirelength, and thus the overall cost. To verify this idea, we invested a random state in ami33 and the distribution of the neighbor solution are shown in Figure 4.2.

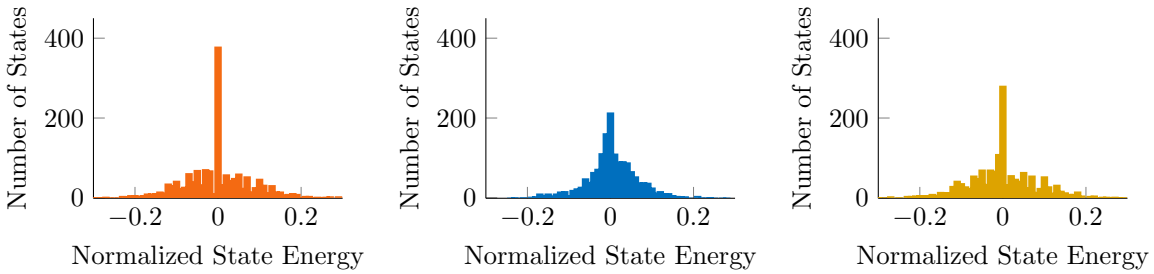


Figure 4.2: Neighbor solution distribution for (a) area minimization; (b) wirelength minimization; (c) area and wirelength minimization.

Table 4.5 shows the results for area and wirelength minimization. Our agent outperforms the

Table 4.5: Area and Wirelength Minimization on MCNC Benchmark. Better results are emphasized in bold.

Circuit	Statistics		Area ($\times 10^6$)		Wirelength ($\times 10^5$)		Cost ($\times 10^6$)		Runtime (s)	
	module #	net #	Ours	SA	Ours	SA	Ours	SA	Ours	SA
apte	9	97	**47.08**	47.31	4.03	**3.43**	**28.41**	28.53	**15.9**	38.1
xerox	10	203	**20.42**	20.64	**6.33**	6.62	**12.51**	12.65	**17.2**	98.8
hp	11	83	**9.21**	9.40	**1.95**	2.62	**5.60**	5.74	**11.6**	44.3
ami33	33	123	**1.24**	1.25	0.69	**0.46**	**0.77**	**0.77**	**43.1**	82.2
ami49	49	408	**38.65**	39.47	**17.24**	12.31	**23.88**	24.18	**66.8**	165.0

simulated annealing algorithm in all the 5 cases.

Table 4.6: Area and Wirelength Minimization on GSRC Benchmark. Better results are emphasized in bold.

Circuit	Statistics		Area ($\times 10^5$)		Wirelength ($\times 10^5$)		Cost ($\times 10^5$)		Runtime (s)	
	module #	net #	Ours	SA	Ours	SA	Ours	SA	Ours	SA
n100	100	576	**1.95**	1.97	1.55	**1.54**	**1.79**	1.80	**389.4**	396.2
n200	200	1274	2.15	**2.01**	3.48	**3.34**	2.68	**2.54**	**784.9**	1101.9
n300	300	1632	3.40	**3.29**	**5.25**	5.44	**4.14**	4.15	3766.9	**2062.3**

4.5.5 GSRC Benchmark

We further conducted experiments on the GSRC benchmark (Caldwell et al., 2002) with larger instances of hundreds of blocks. We tested area and interconnect optimization and listed the results in Table 4.6. Due to the long runtime of the instances, we carefully tune the early-stop criteria on both simulated annealing and our agent. In simulated annealing, the search will finish if no move was accepted in the last 20 temperatures. In our agent, the search will finish if no better solution was found in the last 100 steps.

4.5.6 Result Visualizations and Discussions

Search Progress Visualization

We recorded the search progress of Simulate Annealing and our agent, and visualized them in Figure 4.3. From the figure, we observe that Simulated Annealing explores the action space during searching while our agent searches much smoother. We believe this is because our agent acquires a rather greedy and deterministic heuristic.

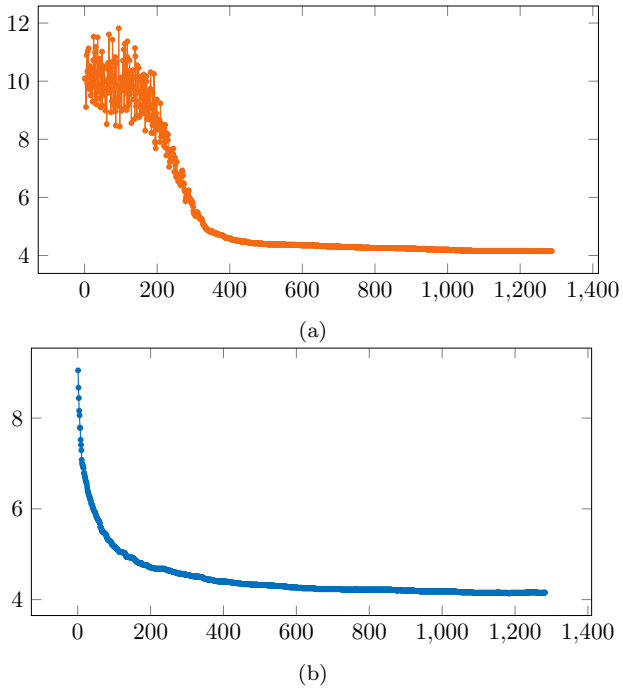
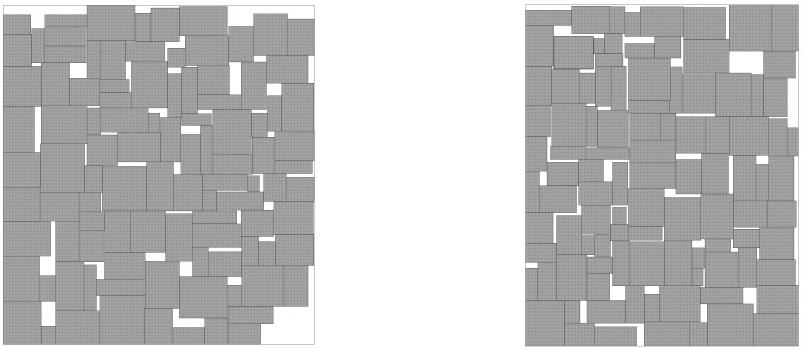


Figure 4.3: Search progress visualizations on the n300 netlist of GSRC benchmark: (a) Simulated Annealing; (b) Our Agent. Our agent searches in a smoother way, indicating a more greedy and deterministic heuristic.

Floorplan Visualization

We visualized the floorplans generated by Simulated Annealing and our agent on the n100 netlist of GSRC Benchmark in Figure 4.4. The dead space of floorplan by our agent is only 7.95% compared to 8.88% dead space by Simulated Annealing.



(a) (b)

Figure 4.4: Floorplan visualizations of the n100 netlist of GSRC benchmark: (a) n100 floorplan by simulated annealing; (b) n100 floorplan by our agent. The dead space of the floorplan by our agent is lower (7.95%) compared to that (8.88%) by Simulated Annealing.

Runtime Profiling

Where is the runtime spent? Instead of roughly showing a total runtime, we profiled both our agent and the simulated annealing algorithm to make better comparisons. Profiling results are in Figure 4.5. According to the profiling, more than half of the runtime (63.1% and 52.3%, respectively)

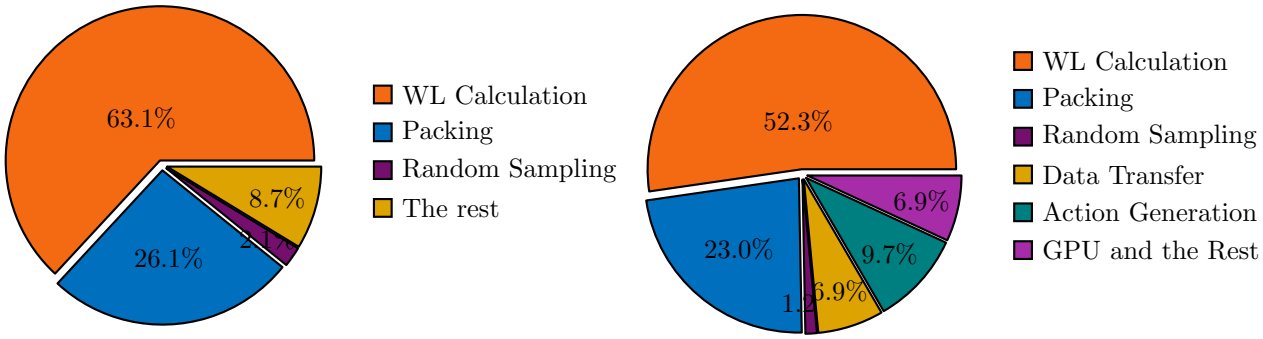


Figure 4.5: Runtime profiling of the Simulated Annealing algorithm (left) and our agent (right). Most of the runtime (89.2% and 75.3%, respectively) is spent on solution evaluation (wirelength calculation and packing).

are spent on wirelength calculation. Packing is the second largest consumer (26.1% and 23.0%), and in other words solution evaluation takes most (89.2% and 75.3%, respectively) of the runtime. Random sampling only takes a little portion of time (2.1% and 1.2%).

4.6 Summary

In this chapter, we investigated the possibility of leveraging reinforcement learning to acquire a floorplanner. The key motivation of our work is to 'learn' new algorithms for difficult combinatorial problems without human expert knowledge. Specifically, we explored the possibility of acquiring local search heuristics through numerous search experiments. To illustrate the applicability, an agent has been trained to perform a walk in the search space by selecting a candidate neighbor solution at each step. We trained the agent using a novel deep Q-learning algorithm with action sampling, and the experimental results have demonstrated the effectiveness of our proposed methods. This work represents the first systematic attempt on leveraging the idea of reinforcement learning to floorplanning.

Chapter 5

Physical Design Optimization for Neural Network Processors

5.1 Introduction

Deep learning has emerged as one of the most important workloads due to its extraordinary performance gains in a number of disciplines. Despite of that, how to efficiently execute deep neural network (DNN) models remains a crucial concern since the beginning of deep learning resurgence. Exacerbating the problem is the progressively sophisticated network architectures, as well as the irregularity due to network pruning and compression, which unarguably impedes the deployment of modern DNNs onto devices. The above challenge intrinsically highlights the demand for customized physical synthesis methodologies, since the quality of physical synthesis directly determine the performance of neural network processors.

As neural network processors getting increasingly hierarchical and structural, dataflow optimization becomes essential to boost the system capabilities. Dataflow optimization schedules operation by data availability, which exposes opportunity for parallelism and data reuse. To illustrate some achievements in dataflow optimization, Zhang et al. (2015) quantitatively analyze the computing throughput and required memory bandwidth using various conventional optimization techniques,

such as loop tiling and transformation, and then apply roofline model to balance the resource; Alwani et al. (2016) fuse the processing of adjacent convolutional layers with the data on-chip and use a pyramid-shaped multi-layer sliding window to minimize off-chip transfer; Ma et al. (2017) present an in-depth analysis of convolution loop acceleration strategy by numerically characterizing the loop optimization techniques and then use multiple optimization algorithms to optimize the loop operation and the dataflow. Zhang et al. (2018) propose a fine-grained layer-based pipeline architecture and a column-based cache scheme for higher throughput, lower pipeline latency, and smaller on-chip memory consumption; Wei et al. (2017) provide an analytical model for performance and resource utilization and develop an automatic design space exploration framework to generate a convolutional neural network (CNN) implementation using systolic arrays. Sun et al. (2019) improve the power performance by combining layer fusion and dataflow optimization techniques.

In addition to optimizing the dataflow itself, dataflow regularity also give rise to new methodologies in physical synthesis, the critical stage in modern very-large-scale integrated (VLSI) circuit design flow. In VLSI design, datapaths are characterised by high degree of bit-wise parallelism, which are placed with high regularity and compactness to achieve high performance (Wang and Shin, 2017). In this sense, datapath-driven placement approaches have been attracting researchers' attention.

5.2 Preliminaries

5.2.1 Neural Network Processor

In deep neural networks, executing inference such as convolution performs a very large amount of multiply-accumulate (MAC) operations, since a single convolution comprises of iterating over every channel and every pixel for each given input, typically with billions or even trillions of iterations. Besides, the model itself must be executed once for each new input.

While central processing units (CPUs) are effective at processing highly serialized instruction streams, machine learning workloads tend to be highly parallelizable, which is a good fit for graphics processing units (GPUs). Moreover, neural processing units (NPUs) benefit from vastly simpler logic because their workloads tend to exhibit high regularity in the computational patterns of deep neural

networks. For the above reasons, many customized dedicated neural processors have been developed.

An NPU is a well-partitioned circuit that includes all the control and arithmetic logic components necessary to execute machine learning algorithms. NPUs are designed to accelerate the performance of common machine learning tasks such as image classification, machine translation, object detection, and various other predictive models. NPUs might be parts of a large SoC, a plurality of NPUs may be instantiated on a single chip, or they may be part of a dedicated neural-network accelerator.

5.2.2 Physical Design Flow

Physical design is based on a netlist synthesized from an RTL design to a gate-level description. Generally, the physical design flow is divided into several steps: floorplanning, partitioning, placement, clock-tree synthesis, routing, physical verification, and layout post-processing with mask data generation. Floorplanning, placement, and routing are the most essential steps in physical design.

Floorplanning determines geometric relations between modules to optimize some objectives such as area, wirelength, and some desired performance. A bad floorplan leads to wastage of die area and routing congestion. As for the circuit performance, lower area is usually desired, as it indicates shorter interconnect distances, fewer routing resources used, faster end-to-end signal paths, and even faster and more consistent place and route time. However, routing may be more difficult with fewer assigned routing resources. In general, floorplanning benefits from hierarchy information like datapaths.

Placement is another crucial stage in physical design. A poor placement not only affects the chip performance but also makes it non-manufacturable with an excessive wirelength beyond available routing resources. Therefore, placement always processes with several objectives to ensure that a circuit meets its performance demands. Routing builds on placement and it assigns wires to properly connect the placed components under all design rules for the integrated circuits. Together, the placement and routing steps of integrated circuits design are known as place and route (PnR).

5.3 Problem Formulation

In the rest of the chapter, we focus on a wafer-scale deep learning accelerator placement problem introduced in ISPD2020 contest (James et al., 2020) as a case study. We argue that the problem is more like a typical floorplanning problem, in which there are tens or hundreds of macro blocks, and the block shapes are flexible (we actually need to determine the shapes in the solution). Therefore, the contest has put forward a floorplanning problem for neural network optimization on a wafer-scale computing engine. Considering that neural networks are a stack of single layers where each performs a single function, naturally an AI compiler decomposes such neural network computation into a stack of single units called *kernels*. To solve this floorplanning problem, we are required to assign each compute kernel a two-dimensional position on the wafer without overlap and utilize computing resources as much as possible.

5.3.1 Kernel Description

Formally, a *kernel* is a parametric program that performs specific tensor operations. For instance, a convolution kernel performs several kinds of convolution operations. To better describe how a kernel is customized on the wafer, its arguments are classified into two main groups. *Formal arguments* specify the exact shapes of tensor operations to be performed. They are uniquely determined by the network architecture, so we consider them to be fixed in our optimization. *Execution arguments* describe how the operation is parallelized across tiles. We are required to find the optimal combination of execution arguments to obtain a maximum utilization of computation resources.

Still, we take a convolution kernel as an example. A normal convolution kernel contains 8 formal arguments, represented by a tuple (H, W, R, S, C, K, T, U). In detail, (H, W) specify the height and width of input image respectively; (C, K) represent the number of channels of input and output image; (R, S) describe the kernel size in two dimensions; and (T, U), similarly, are strides in two dimensions, respectively. They are fixed arguments or intrinsic attributes of a convolution kernel. On the other hand, four execution arguments (h', w', c', k') are free to be specified to maximize resource utilization.

5.3.2 Evaluation

Since we are provided with different types of kernels, each type of kernels have a function to evaluate performance. The performance of a convolution kernel is evaluated by a 4-tuple (h, w, t, m) called *performance cuboid*, where h, w, t, m represent height, width, time and memory this kernel requires, respectively.

The ISPD2020 contest benchmarks provide two main categories of kernels, convolution kernels and block kernels. The performance evaluator of a convolution kernel K (denoted type `conv` hereafter) with formal arguments $\mu_K = (H, W, R, S, C, K, T, U)$ is defined as a function convperf that maps execution arguments $x_K = (h', w', c', k')$ to a resource cuboid $r_K = (h, w, t, m)$, where

$$\begin{aligned} h &= h'w'(c'+1), \qquad w = 3k', \\ t &= \left\lceil \frac{H}{h'} \right\rceil \cdot \left\lceil \frac{W}{w'} \right\rceil \cdot \left\lceil \frac{C}{c'} \right\rceil \cdot \left\lceil \frac{K}{k'} \right\rceil \cdot \frac{RS}{T^2}, \\ m &= RS\frac{C}{c'}\frac{K}{k'} + \frac{W+S-1}{w'}\frac{H+R-1}{h'}\frac{K}{k'}. \end{aligned} \tag{5.1}$$

Notation $\lceil \cdot \rceil$ represents math `ceil` function. Each of block kernel consists of several `conv` kernels. For example, a block kernel K, with formal arguments $\mu_K = (H, W, F)$ and execution arguments $x_K = (h', w', c'_1, \cdots, c'_n, k'_1, \cdots, k'_n)$, comprises of n `conv` kernels $K_i (i = 1, \cdots, n)$. Specifically, a convolution kernel K_i contains $x_{K_i} = (h', w', c'_i, k'_i)$ as its execution arguments. The formal argument tuple μ_{K_i} of `conv` kernel K_i is determined by $\mu_K = (H, W, F)$ and specific attributes of the current block type in detail. The ISPD2020 contest benchmarks provide us with two types of block kernels, `dblock` and `cblock`.

- `dblock`. A `dblock` kernel consists of 3 different `conv` kernels K_1, K_2, K_3. Corresponding formal arguments are

$$\begin{aligned} \mu_{K_1} &= (H, W, 1, 1, \quad F, F/4, 1, 1), \\ \mu_{K_2} &= (H, W, 3, 3, F/4, F/4, 1, 1), \\ \mu_{K_3} &= (H, W, 1, 1, F/4, \quad F, 1, 1). \end{aligned}$$

- `cblock`. A `cblock` kernel consists of 4 different `conv` kernels K_1, K_2, K_3, K_4. Similarly the

corresponding formal arguments are

$$\begin{aligned}
\mu_{K_1} &= (\quad H, \quad W, 1, 1, F/2, F/4, 1, 1),\\
\mu_{K_2} &= (\quad H, \quad W, 3, 3, F/4, F/4, 2, 2),\\
\mu_{K_3} &= (H/2, W/2, 1, 1, F/4, \quad F, 1, 1),\\
\mu_{K_4} &= (\quad H, \quad W, 1, 1, F/2, \quad F, 2, 2).
\end{aligned}$$

Formal arguments of each component `conv` kernel are uniquely determined by $\mu_K = (H, W, F)$. Similar to the performance evaluator of `conv` kernels, the performance of a block kernel K consisting of n different `conv` kernels $K_i (i = 1, \cdots, n)$ can be evaluated by

$$\text{blockperf}(\mu_K; x_K) = r_K = (h, w, t, m), \tag{5.2}$$

where components of 4-tuple r_K is formulated as

$$\begin{aligned}
h = \max_{1 \le i \le n} h_i, \quad w = \sum_{i=1}^{n} w_i,\\
t = \max_{1 \le i \le n} t_i, \quad m = \max_{1 \le i \le n} m_i;
\end{aligned} \tag{5.3}$$

$$(h_i, w_i, t_i, m_i) := \text{convperf}(\mu_{K_i}; x_{K_i}). \tag{5.4}$$

To formulate the optimization problem, objective and constraints must be reasonably specified. Suppose that we are provided with kernel library $\mathcal{K} = \{K_1, K_2, \cdots, K_n\}$, where K_i is a kernel of type `conv`, `dblock` or `cblock`. A floorplan solution assigns location vectors $\boldsymbol{x}, \boldsymbol{y}$ to these kernels. Then our final objective function contains three parts:

- **Maximum execution time** of any placed kernel. Since any kernel K in the kernel library $\mathcal{K}$ will be evaluated and return a resource cuboid r_K containing execution time t_K, this term is normally formulated as $\max_{1 \le i \le n} t_{K_i}$.
- **Total L_1 wirelength**. It is determined by a floorplan $\boldsymbol{x}, \boldsymbol{y}$ of kernels $\mathcal{K}$. We formulate total wirelength as such a function $W(\boldsymbol{x}, \boldsymbol{y})$ of location vectors.
- **Protocol differences** P between connected kernels.

Suppose that there is a directed edge from kernel K_1 to kernel K_2, whose execution arguments are

represented by

$$x_{K_1} = (h'^{(1)}, w'^{(1)}, c_1'^{(1)}, \cdots, c_m'^{(1)}, k_1'^{(1)}, \cdots, k_m'^{(1)}),$$

$$x_{K_2} = (h'^{(2)}, w'^{(2)}, c_1'^{(2)}, \cdots, c_n'^{(2)}, k_1'^{(2)}, \cdots, k_n'^{(2)}).$$

Note that when m or n is exactly 1, the kernel type is `conv`. Then the protocol number of this edge is defined as

$$P = 3 - \delta_{h'^{(1)}, h'^{(2)}} - \delta_{w'^{(1)}, w'^{(2)}} - \delta_{p^{(1)}, p^{(2)}}, \tag{5.5}$$

where $p_1 = \min\{c_m'^{(1)}, k_m'^{(1)}\}$, $p_2 = \min\{c_1'^{(2)}, k_1'^{(2)}\}$ are split numbers in two dimensions, and δ_{ij} is *Kronecker delta* which takes value 1 if the two subscripts are equal and 0 otherwise. The protocol number is uniquely determined by the kernel graph G.

The objective we are about to optimize is the weighted sum of the three parts mentioned above.

$$J = \max_{1 \le i \le n} t_{K_i} + \lambda_1 W(\boldsymbol{x}, \boldsymbol{y}) + \lambda_2 P(G), \tag{5.6}$$

where λ_1 and λ_2 are hyper-parameters provided by benchmarks. J should be optimized under specific constraints, 1) all kernels must fit within the fabric area; and 2) kernels must not overlap. The constraints are straight-forward for industrial practitioners to follow. They are discrete enough to make our optimization problem difficult to solve.

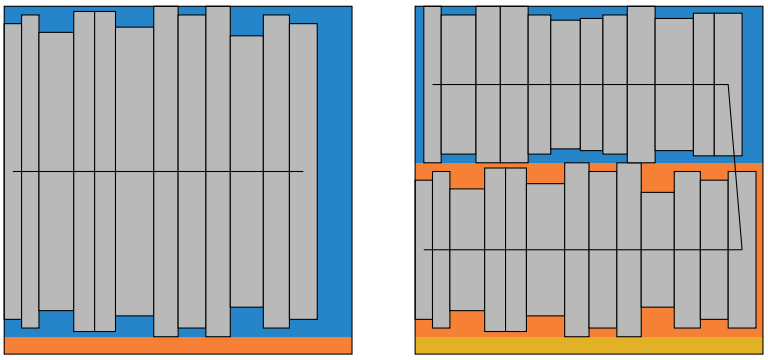
5.4 Algorithm

Considering that a neural network usually is a stack of layers, we can extract a clear datapath to describe how data is processed during a forward pass. That inspires us that it is possible to arrange the floorplan according to the datapath, since we have great flexibility to control the shape of each module.

5.4.1 One-Row Floorplan

Many cases in ISPD2020 contest benchmarks are a chain-like connection of kernels. Intuitively, if the total number of kernels is not very large, we can always place them horizontally one by one following the dataflow order, shown in Figure 5.1a.

Take a look at the performance function of a `conv` kernel. If we ignore the math `ceil` function,



(a) (b)

Figure 5.1: (a) One-row floorplan; (b) Multi-row Mamba floorplan.

it can be easily derived that execution time t is inversely proportional to kernel area, approximately. However, three unroll factors h', w', k' are capable of affecting the value of height h, while w is only affected by c'. Therefore, it is more reasonable to pinch each module to a thin and tall one, so that the height can shoulder more unrolling burden.

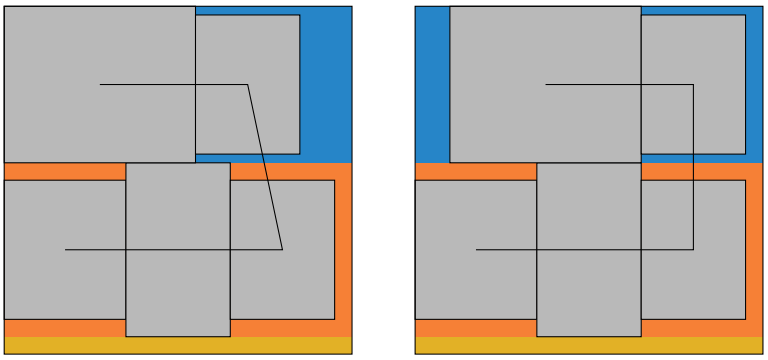
This one-row strategy should work pretty fine especially when the weight of wirelength λ_1 is significantly considerable, since the total Manhattan distance between connected kernels could be tremendously optimized to an observable value.

5.4.2 Mamba Floorplan

Unfortunately, the one-row floorplan described in the subsection 5.4.1 does not always work. In fact, in most of large cases it will definitely fail, considering that the width allowed for floorplan is at most 633 while the minimum width of a block is at least 3. Then it is natural to extend our solution to multi-row floorplan. Specifically, to reduce the total wirelength, it should be better to connect the rows head-to-head and tail-to-tail, and thus we call such floorplan strategy *Mamba* floorplan, illustrated in Figure 5.1b.

5.4.3 Floorplan Compacting

From Figure 5.1b we observe that our mamba floorplan strategy can be further compressed. Once the total width of kernels placed in a row is not strictly equal to the maximum wirelength, we can



(a) (b)

Figure 5.2: (a) Mamba floorplan; (b) Horizontally compacted Mamba floorplan.

always slide them horizontally preserving the kernel order. Figure 5.2 illustrates our compacting strategy. In Figure 5.2a, the blue and orange colored regions are the first and second row respectively, and the black lines connecting five kernels indicate the datapath.

We observe that neither the two kernels in the first row nor the three kernels in the second row are able to fill the width of corresponding row without any gap. Therefore we can horizontally move kernels preserving order to further reduce the total wirelength, shown in Figure 5.2b. The abstracted wire connecting the two rightmost kernels are strictly vertical so that the total wirelength is guaranteed to be less than that in Figure 5.2a.

In general cases where we apply multi-row mamba floorplan, the total number of rows might be larger. Therefore our row-shift will be much more complicated. Assume that we have n rows in total. For the i-th row, we have three variables a_i, b_i, r_i determining its characteristics, where a_i is the *center* horizontal coordinates of the leftmost kernel, b_i is the *center* horizontal coordinates of the rightmost kernel, and r_i is the total width of all consecutive kernels placed in this row. The goal of compacting is to find $x_1, \cdots, x_n$ where x_i represents the distance between the left boundary of region

and the leftmost kernel of the i-th row. This problem can be formulated as a linear programming.

$$\begin{aligned}
\min \ & \sum_{i=1}^{n-1} |x_i - x_{i+1} + t_i| \\
\text{s.t.} \ & x_i + r_i \leq w, \ \ i = 1, \cdots, n, \\
& x_i \geq 0, \ \ i = 1, \cdots, n, \\
& t_{2i} = a_{2i} - a_{2i+1}, \ \ i = 1, \cdots, \left\lfloor \frac{n-1}{2} \right\rfloor, \\
& t_{2i-1} = b_{2i-1} - b_{2i}, \ \ i = 1, \cdots, \left\lfloor \frac{n}{2} \right\rfloor,
\end{aligned} \tag{5.7}$$

where w represents the total width of the floorplan region. Any solver aiming at solving linear programming can be applied to Problem (5.7). Empirically the total number of rows is always a small number (less than 6) to make the solving efficient.

5.4.4 Execution Arguments Selection

We have already determined the floorplan strategy, so the only thing left is to decide the execution arguments of the kernels. Generally, execution arguments of kernels are not independent to each other, however, we tend to consider them separately to reduce the computational complexity.

For a `conv` kernel, we simply perform a grid search in all possible combinations of h', w', c', k'. Note that lots of solutions can be trivially pruned. For example, $c' = \lceil C/2 \rceil$ and $c' = C - 1$ yield the same unroll strategy on C because $\left\lceil \frac{C}{\lceil C/2 \rceil} \right\rceil = \left\lceil \frac{C}{C-1} \right\rceil$, while the latter one consumes far more processing tiles when C is large. Specifically, we have the following results.

Lemma 2. *For a positive integer N, there are $\left\lfloor \sqrt{N} \right\rfloor$ different integer pairs $(x, \lceil \frac{N}{x} \rceil)$ such that $1 \leq x \leq \lceil \frac{N}{x} \rceil$ and $\lceil \frac{N}{x} \rceil \geq 1$.*

Theorem 6. *For any positive integer N, there are at most $2\left\lfloor \sqrt{N} \right\rfloor$ different integer pairs $(x, \lceil \frac{N}{x} \rceil)$ such that $x \geq 1$ and $\lceil \frac{N}{x} \rceil \geq 1$.*

Proof. From Lemma 2 we know that $(1, \lceil N \rceil)$, $(2, \lceil N/2 \rceil)$, $\cdots$, $\left(\left\lceil \sqrt{N} \right\rceil, \left\lceil N/\left\lceil \sqrt{N} \right\rceil \right\rceil\right)$ are legal pairs. Take arbitrary positive $x \leq N$, and let $y = \lceil N/x \rceil$. It is obvious that $x = \lceil N/y \rceil$ is also true. Therefore, if (x, y) is a legal pair such that $x > y$ according to theorem description, then (y, x) is

also legal and thus $y \in \{1, \cdots, \left\lceil \sqrt{N} \right\rceil\}$. If N is a square number, $(\sqrt{N}, \sqrt{N})$ is legal, then the total number is $2\left\lfloor \sqrt{N} \right\rfloor - 1$, otherwise it is $2\left\lfloor \sqrt{N} \right\rfloor$. □

Theorem 6 indicates that, we only need to perform a grid search in $\mathcal{O}(\sqrt{HWCK})$ time complexity, providing that if $\lceil H/h'_1 \rceil = \lceil H/h'_2 \rceil$ we always prefer the smaller one $\min\{h'_1, h'_2\}$ to make the `conv` kernel compact.

For a `dblock` or `cblock` kernel, we balance the heights of its internal `conv` kernels first, otherwise the resulted empty space due to the height difference will be wasted. Take a `dblock` kernel as an example. It has three `conv` kernels $K_i (i = 1, 2, 3)$ with execution arguments $x_{K_i} = (h', w', c'_i, k'_i)$. To balance the height of kernels we have $c'_1 = c'_2 = c'_3$. Similarly, to balance the runtime we have $k'_1 : k'_2 : k'_3 = 4 : 9 : 4$, based on the following results.

Theorem 7. *For a `dblock` or `cblock` kernel K, its execution arguments $(h', w', c'_1, \cdots, c'_n, k'_1, \cdots, k'_n)$ is no better than the modified one $(h', w', c', \cdots, c', k'_1, \cdots, k'_n)$ where $c = \max_i\{c'_i\}$ with respect to height, width, time and memory.*

Proof. The proof is straight-forward, since runtime t and memory m is non-increasing with respect to c', and width w is irrelevant to c'. Height is the maximum of three `conv` kernels, from $\max_{i\in\{1,\cdots,n\}}\{h'w'(c'_i+1)\} = h'w'(c'+1)$, we know that the height to this `dblock` remains unchanged. Therefore the performance of execution arguments $(h', w', c', \cdots, c', k'_1, \cdots, k'_n)$ is no worse than the original one. It applies to the `dblock` kernel when $n = 3$, and `cblock` kernel when $n = 4$. □

The result related to k' is not easy to derive because of the `ceil` functions. From the argument selecting strategy for `conv` kernel we tend to select those numbers that *roughtly* divide the corresponding formal arguments as execution arguments, *e.g.* h' such that H/h' is close to $\lceil H/h' \rceil$. Therefore, it should be reasonable to approximate $\lceil H/h' \rceil$ by H/h'. We define the approximated runtime $\tilde{t}$ of a `conv` kernel K with formal arguments $\mu_K = (H, W, R, S, C, K, T, U)$ and execution arguments $x_K = (h', w', c', k')$ as follows,

$$\tilde{t} := \frac{HWCKRS}{h'w'c'k'T^2}. \tag{5.8}$$

For `block` kernels, $\tilde{t}$ is defined as the maximum of that of its `conv` kernels. We have the following result.

Theorem 8. *Given a* `dblock` *kernel* K *with execution arguments* $x_K = (h', w', c', c', c', k'_1, k'_2, k'_3)$, *let* $k' = \min\{k'_1, k'_3\}$. *Then* x_K *is no better than* $(h', w',\ c', c', c', \lceil \frac{4}{9}k'_2 \rceil, k'_2, \lceil \frac{4}{9}k'_2 \rceil)$ *if* $\frac{4}{9}k'_2 \leq k'$, *otherwise no better than* $(h', w',\ c', c', c', k', \lceil \frac{9}{4}k' \rceil, k')$ *with respect to height, width and approximated time defined in Equation* (5.8).

Proof. The height is irrelevant to k' so it remains unchanged. Consider the approximated time.

$$\tilde{t}_1 = \frac{HWF^2}{4h'w'c'k'_1},\ \tilde{t}_2 = \frac{9HWF^2}{16h'w'c'k'_2},\ \tilde{t}_3 = \frac{HWF^2}{4h'w'c'k'_3}.$$

Suppose that $\frac{4}{9}k'_2 \leq k'$, then $\tilde{t}_2 \geq \max\{\tilde{t}_1, \tilde{t}_3\}$ and thus $\tilde{t}$ is not determined by k'_1, k'_3. We simply decrease k'_1 and k'_3 such that they are still no less than $\frac{4}{9}k'_2$, then $\tilde{t}$ of this kernel must remain unchanged and additionally total width is reduced.

If $\frac{4}{9}k'_2 > k'$, then $\tilde{t} = \max\{\tilde{t}_1, \tilde{t}_3\}$ and thus $\tilde{t}$ is determined by k'_1, k'_3. We decrease k'_1 and k'_3 to k', and let k'_2 be the minimum number such that $\frac{4}{9}k'_2 \geq k'$ (in other words $k'_2 = \lceil \frac{9}{4} \rceil k'$), it is clearly that $\tilde{t}$ remains unchanged but $w = 4(k'_1 + k'_2 + k'_3)$ is reduced. Hence we completed the proof. □

Theorem 8 indicates that the optimal settings of `dblock` kernel execution arguments should have $k'_1 : k'_2 : k'_3 \approx 4 : 9 : 4$. Similarly, in `cblock` kernels, execution arguments $k'_1 : k'_2 : k'_3 : k'_4 \approx 8 : 9 : 4 : 8$ are reasonably close to optimum. In our argument selecting strategy, such prior knowledge significantly reduces the search space.

5.5 Experimental Results

We implemented our `cupid` kernel floorplan engine in `C++` programming language on a 64-bit Linux machine with a 3.4GHz Intel Xeon CPU and 32GB RAM. The results are evaluated on the ISPD20 contest benchmark suites (James et al., 2020).

The benchmark statistics and our experimental results are listed in Table 5.1. Weights λ_1 and λ_2 represent the weight of wirelength and protocol cost in the objective function, respectively. The `cost` columns describe the value of the objective in Equation (5.6). Our methodology performs much better than the baseline floorplan algorithm (Jiang et al., 2020) based on twin binary sequence (TBS) (Young et al., 2003) and simulated annealing, although the latter one could handle more general cases.

Table 5.1: Benchmark statistics and experimental results.

Benchmark Statistics				TBS SA Algorithm (Jiang et al., 2020)				Our Strategy			
Case	#Kernels	λ_1	λ_2	Max Time	WL	Protocol	Cost	Max Time	WL	Protocol	Cost
A	17	1	0	37044	3611.5	11	40655.5	35280	2047	13	**37327**
B	34	1	0	70560	6657	20	77217	65856	4905	17	**70761**
C	102	1	0	76608	15696	69	92034	65772	4278	281	**70050**
D	54	1	0	38304	9327.5	44	47631.5	34944	3071.5	89	**38015.5**
E	17	10	100	36288	2080.5	7	57793	39690	590	16	**47190**
F	34	10	100	76608	3237	15	110478	70560	1475	14	**86710**
G	102	10	100	91728	7784	29	172468	69888	2508	141	**109068**
H	54	10	100	47040	4450	21	93640	43008	893	115	**63438**
I	27	4	0	56448	3790	16	71608	52920	612	13	**55368**
J	81	4	0	63504	8009.5	52	95542	57792	1117.5	286	**62262**
K	18	4	0	576	236	3	**1520**	504	400	14	2104
L	54	4	0	1280	910.5	60	4922	504	774	114	**3600**
M	25	4	0	2359296	9359	24	2396732	2336256	5100	67	**2356656**
N	28	4	0	2268	707.5	0	5098	1599	448.5	9	**3393**
O	27	40	400	63504	1202	6	113984	52920	612	13	**82600**
P	81	40	400	115101	4015	24	285301	66528	2273	102	**198248**
Q	18	40	400	1152	178	1	**8672**	504	400	14	22104
R	54	40	400	1372	1443	30	**71092**	504	774	114	77064
S	25	40	400	2495376	3551	25	2647416	2396160	1899.5	65	**2498140**
T	28	40	400	5720	555.5	0	27940	2015	367.5	9	**20315**
Avg							1.17×				**1.00**×

5.6 Summary

In this chapter, we discussed physical synthesis for advanced neural network processors. Due to the regularity of neural network processors, we argue to utilize datapath driven placement that takes circuit topology into physical design consideration. We scrutinized a wafer-scale deep learning accelerator placement problem, a case study of specific physical synthesis for advanced neural network processors. Experimental results show that datapath driven floorplan greatly outperforms standard methods such as simulated annealing.

Chapter 6

GPU-Accelerated Design Rule Checking

6.1 Motivation

Design rule checking (DRC) is a critical stage in VLSI design flow that ensures a layout satisfies a deck of design rules imposed by process technology. Modern design rules consist of complex geometric constraints, such as constraints on distance, area, alignment, shape, and so on. Moreover, these rules may involve interactions between layers (e.g., constraints on the `NOT CUT` result between layers, minimum overlapping area constraints), as well as conditional rules (e.g., different spacing constraints given different projection lengths). Recent advancements in process technology have also significantly impacted design rule checking. A practical impact is an explosion in the number of design rules that must be honored in the layout. The facts above have pushed DRC to become one of the most time-consuming stages in the whole design flow.

Despite the fact that various parallel DRC algorithms have been investigated (introduced in Section 2.4), their scalability still cannot catch up with the growth of computation demand of DRC for modern designs under advanced processes. It takes more than a day and more than 2000 cores

to complete an entire DRC on a 5nm design[1]. To achieve reasonable design cycle time, further acceleration for computationally intensive DRC tasks has been demanded to accommodate the ever-growing complexity of modern VLSI circuits. We argue that it is practical to orchestra novel DRC algorithms with modern parallel compute substrate.

Meanwhile, open-source EDA tools have been inspiring and empowering the evolution of cutting-edge EDA research. Many remarkable research outcomes would not be possible without the existence of public pioneering tools ABC (Brayton and Mishchenko, 2010), FLUTE (Chu and Wong, 2008), OpenTimer (Huang and Wong, 2015), DreamPlace (Lin et al., 2019), OpenROAD (Ajayi et al., 2019), etc. An open-source EDA tool facilitates researchers by 1) offering an off-the-shelf working solution to complete specific tasks, 2) serving as a strong baseline for algorithm development, and 3) providing infrastructures for data collection and golden result acquiring for ML applications. When it comes to 'design rule checking' in the literature, detailed routers (e.g., TritonRoute (Kahng et al., 2018)) and layout editors (e.g., Magic (Ousterhout et al., 1985), KLayout (Köfferlein, 2018)) often integrate design rule checkers. Although basic design rule checking algorithms are implemented within these tools, they are not designed solely for physical verification purposes: detailed routers handle fundamental 'design rules' like short, spacing, and minimum area (Chen et al., 2020), while they are tightly coupled with the path search algorithms; layout viewers/editors are graphical user interface (GUI) centric, which are not optimized for standalone design rule checking. As design rule checking is a critical stage where many interesting research and design problems remain unsolved, we feel that a new design rule checking engine is necessary to support all these explorations.

6.2 Preliminaries

6.2.1 Design Rules

Design rules consist of geometric constraints imposed by specific fabrication technologies to achieve a high yield. Distance constraints are the most common constraints, which include, depending on the positional relation between objects, width rules, spacing rules, extension rules, enclosure

[1]Statistics are provided by an industrial partner. The source is omitted due to confidentiality.

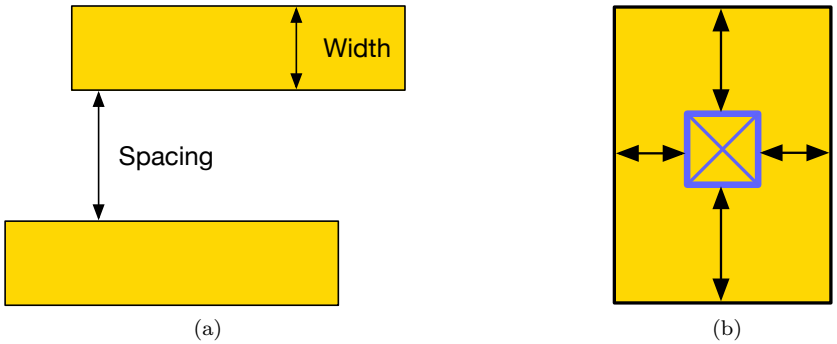


Figure 6.1: Typical rules: (a) *width* and *spacing* rules in a metal layer; (b) *enclosing* rule between a metal layer and a via layer.

rules, and so on. Distance rules usually require a *minimum* distance between polygon edges due to various reasons (Bhanushali and Davis, 2015): the minimum width of polygons is limited by the resolution of the lithographic technique used, the minimum spacing between polygons is to ensure electrical isolation, and the minimum enclosure is to avoid layer misalignment errors. Other popular rules are *minimum area* rules, *shape* constraints (e.g., rectilinear), and *multi-color* design rules for multi-patterning lithography.

We illustrate a few fundamental intra-layer and inter-layer rules. Intra-layer constraints define interactions, measurements, and connectivity requirements between objects on the same layer, e.g., minimum dimensions of objects on each layer or minimum spacing between objects on the same layer, as shown in Figure 6.1a. Inter-layer constraints define interactions, measurements, and connectivity requirements between objects on multiple layers, e.g., encapsulation dimensions for objects on different layers or minimum spacing between objects on different layers, as shown in Figure 6.1b.

6.2.2 Parallel Computation Model

In this chapter, we adopt the PRAM model with concurrent reads and exclusive writes (CREW). We use the *work-depth* (WD) paradigm (JéJé, 1992) to analyze parallel algorithms, where *work* W is the total number of operations, and *depth* D is the length of the critical path, assuming infinitive processing resources. By the Brent's principle, the runtime T_p of an algorithm using p processors can be bounded by $T_p \leq W/p + D$.

6.2.3 General-Purpose GPU and CUDA

The prosperous development of artificial intelligence has also popularized the concept of general-purpose graphics processing unit (GPGPU), which runs general-purpose programs on the hardware architecture initially dedicated to graphics rendering. GPGPUs offer massive computing power for highly parallel applications in various disciplines, which finds orders of magnitude performance gain. The programming model for GPGPUs is best described as Single Program Multiple Data (SPMD), where many parallel processing elements execute a single program on different input data, making them a good fit for data parallelism.

To enhance GPU programmability, higher-level programming environments have emerged, such as CUDA (Nickolls et al., 2008), OpenCL (Munshi, 2009), and OpenACC (Wienke et al., 2012). CUDA comes with a software stack that extends C++ as the programming interface, attracting great attention in academia and industry. CUDA offers a thread hierarchy to organize parallel threads, which form one-, two-, or three-dimensional *thread block*s (NVIDIA, 2023). *Thread block*s are similarly organized into *block grid*s. To allocate computation onto the above threads, CUDA defines *kernel* with an execution parameter N, which will be launched N times in N different CUDA threads. Each such thread is given a unique thread ID accessible with built-in variables in the *kernel*.

In a CUDA program, we refer to the CPU as *host* and the GPU as *device*, and both of them maintain separate memory spaces in their own DRAM. In the compilation flow, a host compiler (e.g., GCC) compiles the *host* code into an executable on the host, while NVidia C Compiler (NVCC) cross-compiles the device code (i.e., those qualified by `__device__`) into CUDA binary, which will be handled by the CUDA runtime system whenever it is invoked from the host program.

6.2.4 DRC Engine in KLayout

As we are going to integrate our proposed parallel DRC algorithms into the DRC flow provided by KLayout (Köfferlein, 2018), we introduce the basics of the KLayout DRC engine. The DRC functionality in KLayout is controlled by a DRC script that specifies the check options and steps. KLayout organizes a layout as layers, which are basically collections of polygons or edges. Large layouts are first clipped into tiles to reduce memory requirements and to enable parallel processing

by multiple CPU cores. In each tile, the touched objects are merged into a single object (the so-called *clean* state in KLayout). The checking tasks are then performed on the merged layers.

6.3 X-Check: GPU-Accelerated Design Rule Checking via Parallel Sweepline Algorithms

6.3.1 Design Rule Checking Algorithms

Problem Formulation

Before diving into technical details, we first describe a general *distance check* problem that we aim to solve.

Problem 2 (Distance Check (informal))**.** *A layout can be seen as a set of axis-parallel polygonal objects. The distance rule says the following: any two edges must not be closer than a predefined minimal distance. A distance violation is a pair of edges in the layout that violate the distance rule. Given a layout, the task is to report all the distance violations.*

Without loss of generality, we first consider horizontal segments only. We now give a more formal definition of the above problem:

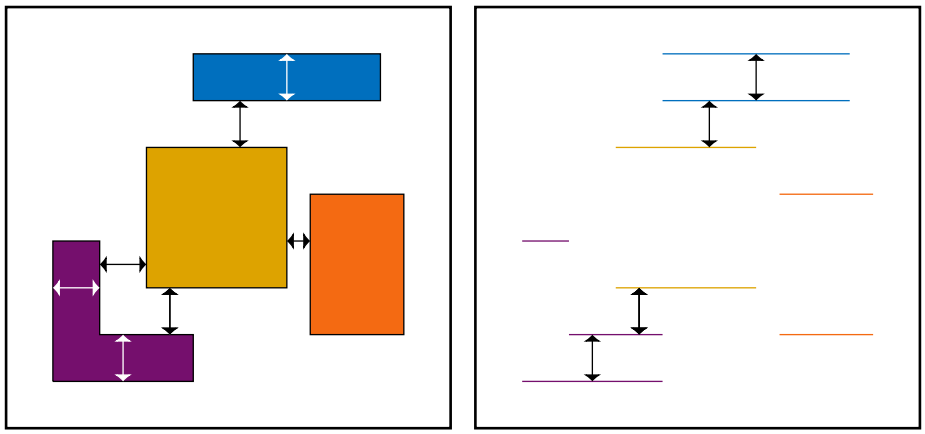
Problem 3 (Distance Check)**.** *Given a set $\mathcal{H}$ of horizontal segments in $\mathbb{R}^2$, report the segment pairs from $\mathcal{H}^2$ whose horizontal projection is nonempty, and vertical distance is smaller than δ. Formally, we want to report:*

$$\begin{aligned} &\{([l_1, r_1] \times y_1, [l_2, r_2] \times y_2) \in \mathcal{H}^2\} \\ &s.t.\ [l_1, r_1] \cap [l_2, r_2] \neq \emptyset, |y_1 - y_2| < \delta \end{aligned} \tag{6.1}$$

Figure 6.2 illustrates the our problem formulation.

Sweepline Algorithms

Technically, Problem 3 can be efficiently solved by the sweepline algorithmic framework. A sweepline algorithm can be conceptually regarded as moving a sweepline on the plane to process a set of points (a.k.a. *event point*s) one by one. Suppose the event points are stored in a data structure $\mathcal{P}$ that



(a) Distance Check (informal) (b) Distance Check

Figure 6.2: Distance Check. See Problems 2 and 3.

supports a *delete-min*[2] operation in $T(\mathcal{P}_{delete-min})$ time. While $\mathcal{P}$ is not empty, the algorithm processes the points p in $\mathcal{P}$ by repeatedly calling the delete-min operation. For each event point p, the algorithm updates a (persistent) status data structure $\mathcal{S}$ that supports insertion, deletion, and range-report, whose time complexities are denoted as $T(\mathcal{S}_{insert})$, $T(\mathcal{S}_{delete})$, and $T(\mathcal{S}_{range-report})$, respectively. The total runtime for the sweepline algorithm can be written as

$$\begin{aligned} T_{total} = |\mathcal{P}|T(\mathcal{P}_{delete-min}) + n \cdot T(\mathcal{S}_{insert}) \\ + n \cdot T(\mathcal{S}_{delete}) + \sum^{m} T(\mathcal{S}_{range-report}), \end{aligned} \tag{6.2}$$

where n is the total number of elements to be inserted/deleted to $\mathcal{S}$, and m is the total number of range-reports.

To solve Problem 3 with a sweepline algorithm, assume all the event points (i.e., endpoints of segments) are known ahead of time. We use a logarithmic time priority queue (e.g., a binary heap) to organize the event points by prioritizing their x-coordinates, which supports delete-min in $O(\log |\mathcal{P}|)$ time. For each event point, we update a self-balancing binary search tree (e.g., a BB[α] tree) that organizes horizontal segments by their y-coordinates, with $T(\mathcal{S}_{insert})$ and $T(\mathcal{S}_{delete})$ in $O(\log |\mathcal{S}|)$ time, and $T(\mathcal{S}_{range-report})$ in $O(\log |\mathcal{S}| + k)$ time where k is the number of reported elements. Specifically, for a left endpoint of a horizontal segment, we insert the segment into $\mathcal{S}$; for a right

[2] A *delete-min* operation finds the minimum element and removes it.

endpoint of a horizontal segment, we remove it from $\mathcal{S}$. When a segment $[l, r] \times y$ is inserted into $\mathcal{S}$, we also query $\mathcal{S}$ and report segments that are within $[y - \delta, y + \delta]$, which all violate the distance rule. Algorithm 2 summarizes the squential sweeping algorithm for distance check. Suppose there are n segments and k violations, we have

$$\begin{aligned} T_{total} &= O(n)T(\mathcal{P}_{delete-min}) + O(n) \cdot T(\mathcal{S}_{insert}) \\ &\quad + O(n) \cdot T(\mathcal{S}_{delete}) + \sum^{O(n)} T(\mathcal{S}_{range-report}) \\ &= O(n \log n + k) \end{aligned}$$

The runtime bound is optimal, as the element uniqueness problem (lower bounded by $\Omega(n \log n)$) is reducible to the problem (Shamos and Hoey, 1976), and we need $\Omega(k)$ time to report all the violations.

Algorithm 2 Sequential Sweepline Algorithm for Distance Check

Require: A set $\mathcal{H}$ of horizontal segments
Ensure: Segment pairs that violate the distance rule
Sort segment endpoints $\mathcal{P}$ by ascending x-coordinates
Initialize an empty BST $\mathcal{S}$ ▷ use y-coordinates as keys
for all endpoint $p \in \mathcal{P}$ **do**
    if p is the left endpoint of a segment $h = [l, r] \times y$ **then**
        Range query $\mathcal{S}$ for $[y - \delta, y + \delta]$
        Report the corresponding segment pairs
        Insert h to $\mathcal{S}$
    else
        Delete h from $\mathcal{S}$
    end if
end for

Parallelizing Sweepline Algorithms

We present a parallel sweepline paradigm proposed by Sun and Blelloch (2019), the key idea of which is to regard a sweepline algorithm as computing prefix structures. We follow the notations used by Sun and Blelloch (2019). Event points $p_i \in P$ are processed in a total order $\prec$: $P \times P \mapsto \{0, 1\}$. At each point, our goal is to build the intermediate data structure $t_i \in T$ with the previous data structure t_{i-1} and the current point p_i using an *update function* $h : T \times P \mapsto T$ (i.e., $t_i = h(t_{i-1}, p_i)$).

The initial prefix structure is t_0. In this way, we define a sweepline algorithm as a five tuple:

$$SW = \{P, \prec, T, t_0, h\}. \tag{6.3}$$

To describe a parallel sweepline algorithm, we further define two operators, a *fold function* $\rho : \langle P \rangle \mapsto T$ that converts a sequence of points to a prefix structure, and a *combine function* $f : T \times T \mapsto T$ that combines/reduces two prefix structures. We require f to be associative. A parallel sweepline paradigm is defined as:

$$PSW = \{P, \prec, T, t_0, h, \rho, f\}. \tag{6.4}$$

The essence of the parallel sweepline algorithm is to make use of the associativity of the *combine function* f. More precisely, repeatedly updating a sequence of points $\langle P \rangle$ into a sequence of prefix structures $\langle T \rangle$ using the *update function* h, is equivalent to first converting the points into (partial) prefix structures, and then combining the partial prefix structures using the *combine function* f. Sun and Blelloch (2019) propose to compute such prefix structures in three steps:

1. **Batching**. The inputs are sorted and evenly split into b blocks. Each thread converts the consecutive n/b points in one block into a partial sum (i.e., prefix) $T'_{kn/b}$ for $k = 1, 2, \cdots, b$ using the *fold function* ρ.

2. **Sweeping**. A single thread is invoked to sweep the b partial sums using the *combine function* f to compute the prefix structures $T_{n/b}, T_{2n/b}, \cdots, T_n$.

3. **Refining**. The rest of the prefix structures are built using the b prefix structures built in the second step. In each block, the points are processed sequentially to update the prefix structures using h. The b blocks can be done in parallel.

Figure 6.3 illustrates the parallel prefix structure build. The runtime complexity of such a strategy is analyzed by Sun and Blelloch (2019), which depends on the complexity of the functions h, ρ, and f. We will do the analysis in Section 6.3.2 within the concrete (DRC) context.

Bootstrapping. Note that each subproblem in the refining step is of the same type as the original problem, so that we can repeatedly apply the same algorithm for each block. Such bootstrapping

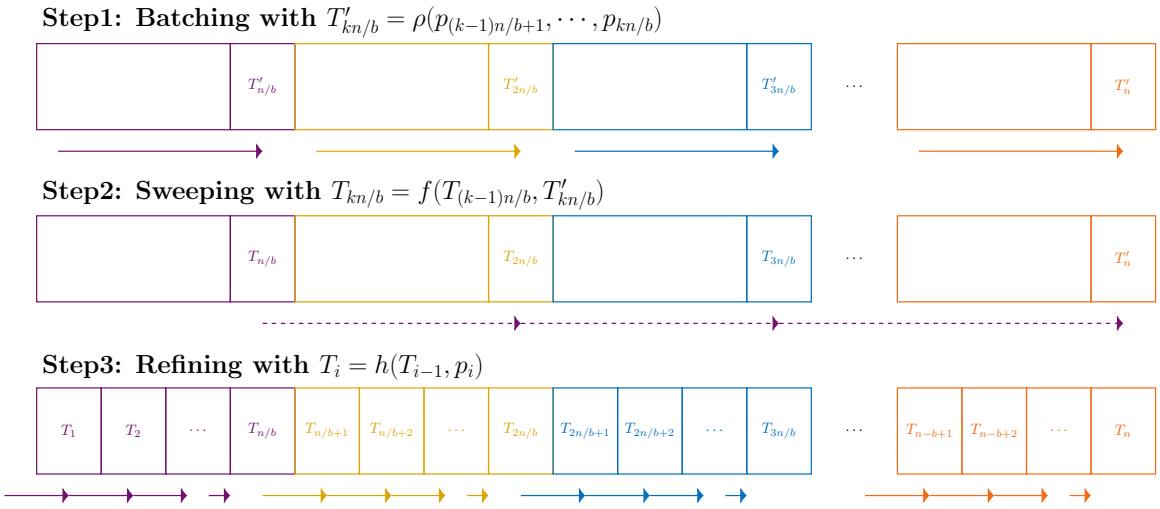


Figure 6.3: Parallel prefix build for sweepline algorithms in three steps: batching, sweeping, and refining. Each rectangle block represents a prefix structure, where different colors indicate different blocks. Each colored arrow represents workload of a thread. Adapted from Sun and Blelloch (2019).

technique may slightly improve the runtime complexity (see Corollary 1 in Sun and Blelloch (2019) for details), usually by some logarithmic factor.

6.3.2 Massively Parallel Design Rule Checking

Section 6.3.1 describes an efficient parallel sweepline algorithmic paradigm. Now, we are going to show that design rule checking tasks fit into the prefix build framework. We claim that distance check (Problem 3) is prefix computation as described in Equation (6.4).

Claim 1. *Distance check is prefix computation.*

To prove the claim, we introduce two strategies to solve the problem, i.e., sweeping vertically and horizontally, in the following subsections.

The Vertical Sweeping Algorithm

Firstly, sort segments in ascending y-coordinates. We explain the algorithm by introducing the components in Equation (6.4).

- The event point set P includes all the y-coordinates of the segments.

- The total order $\prec$ is the total order $<$ on the y-coordinates.
- The prefix structure contains a set $\mathcal{S}$ of segments that are **below current segment within δ in y-direction**.
- The identity t_0 contains an empty set $\emptyset$.
- The *update function* h processes the segments by adding the segment to $\mathcal{S}$, and delete the segments that are below current segment by more than δ.
- For the *fold function* ρ, it suffices to first binary search for the lowest segment that is within δ to the highest segment, and then add the segments in between to the set $\mathcal{S}$.
- The *combine function* f is defined by first taking the union of the sets, and then delete the elements that are below the target segment by more than δ. Note that f is associative because the set operations are associative.

By our construction, the prefix structures contain all the candidate segments below each segment, in the sense that their distances in the y-direction are within δ. It remains to check whether each pair of segments overlap in the x-direction. Each violation will be reported by the algorithm exactly once.

We now analyze the runtime complexity of the vertical sweeping algorithm under the parallel sweepline framework. Recall that we have n events evenly split into b blocks. As an implementation trick, we store all the segments in a global array. The prefix structures only store pointers to this global array instead of explicitly storing the set elements. As a side note, the depth will grow to as large as $O(n \log n)$ if we use a persistent binary search tree for the implementation. We use s_i to denote the size of the i-th prefix structure.

1. **Batching**. There are b blocks, and each block has $O(n/b)$ elements. The ρ function can be implemented using a binary search in the block, which takes $O(\log(n/b))$ time. The total work is $O(b \log(n/b))$.
2. **Sweeping**. Consider the case of combining the prefix structures of the $(k-1)$-th block and the k-th block. After sweeping, the size of $T_{(k-1)n/b}$ is $s_{(k-1)n/b}$, while $T'_{kn/b}$ can have at

most n/b elements. The combine function can be implemented using a binary search in these $s_{(k-1)n/b} + n/b$ elements, which takes $O(\log(s_{(k-1)n/b} + n/b))$ time. Therefore, the total work and depth are $\sum_{k=1}^{b} O(\log(s_{(k-1)n/b} + n/b)$.

3. **Refining**. The b blocks are refined in parallel. In general, building the i-th prefix structure takes $O(\log s_{i-1})$ time. Therefore the total work is $\sum_{k=1}^{n} O(\log(s_{k-1}))$. The depth is bounded by $\max_k \sum_{i=1}^{n/b} O(\log(s_{(k-1)n/b+i-1}))$.

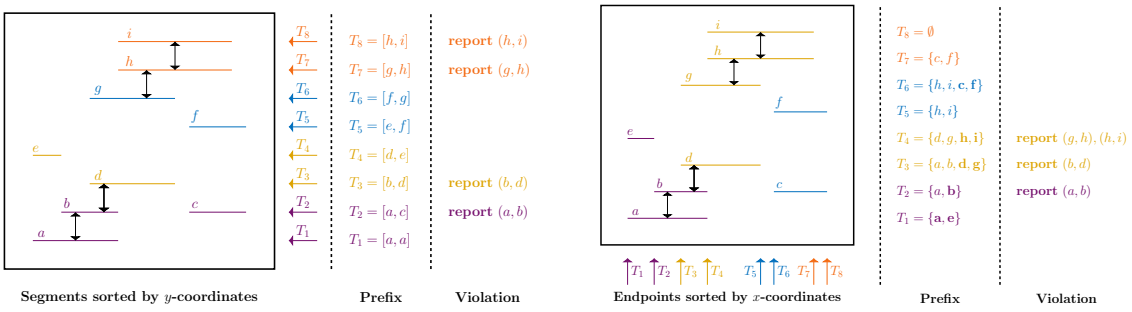
Note that each s_i is upper bounded by i. The prefix structures can be build in $O(n \log n)$ work and $O((b + n/b) \log n)$ depth in the worst case. When $b = \Theta(\sqrt{n})$, the depth is $O(\sqrt{n} \log n)$. This worst-case depth can be obtained by another naive solution that launches b threads to perform the n binary searches in the whole space, resulting in an $O(n \log n/b)$ depth. However, when $s_i = \Theta(\text{polylog}(i))$[3], our algorithm yields the better $O(n \log \log n/b)$ time complexity.

After building the prefix structures, each element in the prefix structures can be examined in constant time for violation check. The total work is bounded by $\sum_{i=1}^{n} s_i$. Again, the worse case complexity is $O(n^2)$, and when $s_i = o(i)$ the algorithm gives nontrivial runtime bound. Algorithm 3 summarizes the vertical sweeping algorithm.

Algorithm 3 Vertical Sweeping

Require: A set $\mathcal{H}$ of horizontal segments
Ensure: Segment pairs that violate the distance rule
1: Sort segments by ascending y-coordinates
2: Partition the sorted segments into b blocks
3: **For** each block **do in parallel** ▷ Batching
4: Find the lowest segment that is within δ to the highest segment in the block
5: **Endfor**
6: Sweep the partial results among the b blocks ▷ Sweep
7: **For** each block **do in parallel** ▷ Refine
8: Refine the prefix structures
9: **Endfor**
10: **For** each prefix structure **do in parallel** ▷ Report
11: Report the violations in the prefix structure
12: **Endfor**

[3]Some literature (Lauther, 1981) gives $\sqrt{n}$ estimation.



(a) Vertical Sweeping (b) Horizontal Sweeping

Figure 6.4: Illustration of vertical and horizontal sweeping algorithms.

The Horizontal Sweeping Algorithm

For horizontal sweeping, we first sort segment endpoints in ascending x-coordinates. We also describe the components according to Equation (6.4).

- The event point set P contains the endpoints of the segments in $\mathcal{H}$.
- $\prec$ is the total order $<$ on the x-coordinates of the endpoints.
- The key observation here is that a sweepline algorithm maintains an 'active set' of segments. This active set of segments are those who span the current (vertical) sweepline, or equivalently, those whose left endpoints are to the left of the sweepline (i.e., have been processed), while the right endpoints are to the right of the sweepline (i.e., have not been processed yet). Therefore, we maintain two sets in the prefix structure $t = (\mathcal{L}, \mathcal{R}) \in T$: a set $\mathcal{L}$ that records the segments whose left endpoints have been processed, and a set $\mathcal{R}$ that records the segments whose right endpoints have been processed. This is natural, as the 'active set' can be easily computed by $L \setminus R$.
- The identity t_0 contains two empty sets, i.e., $t_0 = (\emptyset, \emptyset)$
- The *update function* h processes the endpoint by adding the segment to the corresponding set. Specifically, we have

$$h((\mathcal{L}, \mathcal{R}), p) = \begin{cases} (\mathcal{L} \cup h_p, \mathcal{R}), & \text{if } p \text{ is a left endpoint} \\ (\mathcal{L}, \mathcal{R} \cup h_p), & \text{otherwise,} \end{cases}$$

where h_p is the corresponding segment whose endpoint is p, and $(\mathcal{L}, \mathcal{R})$ is some prefix structure $t \in T$.

- The *fold function* ρ can be trivially defined as applying h for each event point p in a recursive manner. That is:

$$\begin{cases} \rho(p_1) & = h(t_0, p_1) \\ \rho(p_1, p_2, \cdots, p_n) & = h(\rho(p_1, p_2, \cdots, p_{n-1}), p_n) \end{cases}$$

- The *combine function* f is defined using set union, i.e.,

$$f((\mathcal{L}_1, \mathcal{R}_1), (\mathcal{L}_2, \mathcal{R}_2)) = (\mathcal{L}_1 \cup \mathcal{L}_2, \mathcal{R}_1 \cup \mathcal{R}_2)$$

  Note that f is associative because the set union operator $\cup$ is associative.

In this way, we also successfully relate the distance check problem to prefix computation, which can be parallelized using the strategy in Section 6.3.1. We now analyze the time complexity. Assume we organize sets in self-balanced binary search trees (specifically, on their y-coordinates) that support common logarithmic time operations. Merging two binary search trees of sizes m and n takes $O(m + n)$ time.

1. **Batching**. There are b blocks, and each block has $O(n/b)$ elements. The total work is $O(b \cdot n/b \log(n/b))$, and the depth is $O(n/b \log(n/b))$.

2. **Sweeping**. Consider the case of combining the prefix structures of the $(k-1)$-th block and the k-th block. After sweeping, $T_{(k-1)n/b}$ may contain at most $(k-1)n/b$ elements, while $T'_{kn/b}$ can have at most n/b elements. With our assumed tree operation bounds, it costs $\sum_{k=1}^{b} O(kn/b) = O(bn)$ work and the same amount of depth.

3. **Refining**. The b blocks are refined in parallel. Consider the k-th block, where each prefix structure can have at most kn/b elements. Therefore the total work is $\sum_{k=1}^{b} O(n/b \log(kn/b)) = O(n \log n)$. The depth is $O(n/b \log n)$.

By combining the three stages, we have total work $O(n(b + \log n))$ and depth $O(n(b + \log n/b))$. When $b = \Theta(\sqrt{\log n})$, the depth is $O(n\sqrt{\log n})$. This runtime bound is worse than that of the vertical sweeping algorithm, but it maintains the y-coordinates of the segments in order. To report

violations from the prefix structures, it suffices to perform two predecessor/successor searches and report violations within the range, which costs $O(\log n + k)$ work and time, where n is the size of the prefix structure, and k is the number of reported elements. Recall that such a range search takes $O(n)$ work in the vertical sweeping algorithm.

Summary and Discussions

Both algorithms proposed in previous sections give nontrivial runtime guarantees. We summarize them in the following theorem:

Theorem 9. *Assume $s_i = \Theta(\text{polylog}(i))$. Distance check can be solved in $O(n \cdot \text{polylog}(n))$ work and $O(\sqrt{n} \cdot \text{polylog}(n))$ depth, or in $O(n \log n)$ work and $O(n\sqrt{\log n})$ depth.*

We then show that many DRC tasks are distance check.

Claim 2. *Width check is distance check.*

Proof. Let $\mathcal{H}$ be the horizontal segments of a polygon. Let δ be the minimum width constraint. Then, a distance check reports all the horizontal segment pairs that violate the width constraint. Similarly, rotate the polygon by 90°. Now a distance check reports violation between (originally) vertical segments. □

Corollary 1. *Width check can be solved in $O(n \cdot \text{polylog}(n))$ work and $O(\sqrt{n} \cdot \text{polylog}(n))$ depth.*

With almost identical arguments, we have following corollaries.

Corollary 2. *Space check can be done in $O(n \cdot \text{polylog}(n))$ work and $O(\sqrt{n} \cdot \text{polylog}(n))$ depth.*

Corollary 3. *Enclosing check can be done in $O(n \cdot \text{polylog}(n))$ work and $O(\sqrt{n} \cdot \text{polylog}(n))$ depth.*

We also illustrate the vertical and horizontal sweeping algorithms in Figure 6.4. Both vertical and horizontal sweeping algorithms do not dominate each other: they achieve either better work efficiency or better depth bound. From our perspective, the vertical sweeping algorithm appears to be more promising since it provides polynomially better theoretical depth ($\tilde{O}(\sqrt{n})$ v.s. $\tilde{O}(n)$), which is the main reason why we turn to GPU acceleration. Besides, the vertical sweeping algorithm looks easier to implement, as the prefix construction mainly relies on 1D binary search. In contrast,

for horizontal sweeping we have to count on efficient set operations (set union and set difference). This is a bit surprising at the first glance, because we often use a horizontal sweeping strategy for sequential implementation (see Section 6.3.1). We would like to argue that the phenomenon provides an intuitive yet important insight for parallel algorithm design: suppose the problem can be decomposed along both an 'easier' direction and a 'harder' direction, *it is better to decompose a problem by the* 'simple' *direction for parallelism, and leave the* 'complex' *work to each individual processors.* In the distance check case, since the segments are horizontal, the x- and y-coordinates are not equivalent. The y-coordinate is the 'easier' direction because each segment has only one y-coordinate, which forms a total order. This also implicitly enables the use of two pointers to indicate a range (recall how do we represent a prefix structure in the vertical sweeping algorithm). On the contrary, the x-coordinate is the 'harder' one, as each segment has two endpoints with different x-coordinates, and thus there is no global total order of the segments. To deal with the complexity, we proposed to use two sets to maintain the left endpoints and right endpoints separately, but it inevitably complicates the algorithm. Note that the emphasis is different from sequential algorithm design: in the sequential sweepline algorithm, we would like to use the sweepline paradigm to look after the complex x-coordinates and leave the simple y-coordinates to the status data structure $\mathcal{S}$ (defined in Section 6.3.1) that we want to maintain for efficient queries.

6.3.3 GPU Implementation

The massive parallelism exposed in the vertical sweeping algorithm mainly comes from the divide-and-conquer paradigm, which is conceptually GPU-friendly. Whenever the inputs are split into blocks and processed in parallel, we launch multiple GPU kernels to perform the jobs concurrently. Nevertheless, we would like to introduce several implementation details/considerations that we find crucial to obtain satisfying performance.

Dynamic Algorithm Selection

GPU acceleration is not a free lunch. To run applications on GPUs, we inevitably have to move input data from host memory to device memory, launch GPU kernels, wait for synchronization, and move results back from device memory to the host memory. These operations have overhead. When

the degree of parallelism is not high enough to make full utilization of GPU threads, the overhead might dominate the overall runtime and decelerates the whole program.

One straightforward way to compensate for such overhead is to make a dynamic decision of whether executing on GPU helps. For design rule checking, our experience is to estimate the parallelism degree by counting the average number of edges per polygon. The more edges there are in a polygon, the higher chance it has to gain performance from GPU acceleration. Accordingly, we develop a simple dynamic algorithm selection strategy that first calculates the average number of edges per polygon for each tile. If the number is higher than a threshold, we send it to the GPU branch for parallel checking; otherwise, we simply run a sequential checking (i.e., CPU branch) for the tile. The strategy is simple yet effective, as it helps X-Check to match the efficiency of CPU checkers for small/simple tasks. The detailed comparisons are shown in Section 6.3.4.

Sorting Strategy Selection

Sorting is an essential step in our sweeping algorithm. One of the available efficient sorting procedures comes from the *thrust* library, i.e., `thrust::sort`. In the actual program, we need to sort an array of `struct`s by the desired keys, which are some specific fields in the `struct`s. For example, when the `struct`s represent edges, we might want to sort them by the x-coordinates for the vertical edges, and by y-coordinates for the horizontal edges (recall how do we sweep them in the algorithm). The default way to implement is to pass a *comparison function object* as an argument to the `thrust::sort` function. Specifically, `thrust::sort` runs a *merge sort* procedure for such use cases.

Internally, *thrust* also provides a *radix sort* procedure that works for numeric data types (e.g., `int`) and default comparators. Therefore, an alternative solution is to copy the keys out, sort the keys using *radix sort*, and permute the `struct`s according to the sorted results. We call such a solution a Copy-Sort-Permute (CSP) strategy. The code snippet to implement the CSP strategy with *thrust* procedures is shown in Listing 1.

Intuitively, the CSP strategy should only be used with long arrays because it definitely involves more steps and extra work, which would not be desired if sorting itself is already fast enough. In our practice, we only use CSP for arrays of size larger than 8000. Detailed comparisons and more experimental evaluations are shown in Section 6.3.4.

Listing 1 Copy-Sort-Permute to sort long arrays.

```
template <typename S>
void sort_long_arrays(S *array, int n) {
  int *keys;    // the buffer for keys
  int *indices; // the buffer for indices
  S *tmp;       // the buffer for permutation
  // step 0: properly allocate the buffers
  cudaMallocManaged(...)...
  // step 1: Copy
  for (int i = 0; i < n; ++i) {
    keys[i] = array[i].key;
    indices[i] = i;
  }
  // step 2: Sort
  thrust::sort_by_key(keys, keys+n, indices);
  // step 3: Permute
  thrust::copy_n(
      thrust::make_permutation_iterator(
          array, indices),
      n, tmp);
  thrust::copy_n(tmp, n, array);
}
```

Kernel Granularity

It is possible to allocate GPU threads at various granularities; that is, each GPU thread can be responsible for solving a subproblem of various scales. After parallel prefix computation, the primary decision we face is how to report violations from the prefix structures and assign those computational tasks to GPU threads. From coarser-grained to finer-grained, we might assign GPU threads for 1) tile-wise tasks, 2) polygon-wise tasks, 3) tasks indicated by a prefix structure, and 4) a single violation examination task. To allow adequate parallelism, option 1) might not be a good choice. To balance the workload of each thread, options 2) and 3) might not be desired, as the sizes of polygons differ significantly, and the sizes of prefix structures may vary by $\Theta(n)$ in the extreme case, where n is the number of segments in the problem input. Therefore, we decided to implement option 4) in our practice, where we use the unique thread id as the *global* offset to locate its task. Specifically, for the t-th thread, its task is the q-th task in prefix T_p, such that $t = \sum_{i=0}^{p-1} |T_i| + q$.

Table 6.1: Runtime Comparisons of *Width* Check

Design	Layer	#Tiles	#Polygons	#Edges	#Edge/Polygon	Width Check Time (s)		
						KLayout	X-Check	Speedup
gcd	Metal1	1	391	24440	62.5	<0.1	0.1	-
	Metal2	1	1229	4916	4.0	<0.1	<0.1	-
aes	Metal1	16	17739	2059906	116.1	2.9	3.0	0.97×
	Metal2	16	76007	304028	4.0	0.2	0.1	-
bp_be	Metal1	56	34747	27245522	784.1	21.9	19.3	1.13×
	Metal2	56	393834	1575336	4.0	0.4	0.4	-
bp	Metal1	144	107706	52595418	488.3	38.9	33.0	1.18×
	Metal2	144	833588	3334352	4.0	0.9	0.9	-
Average								1.09×

6.3.4 Experimental Results

We implement our algorithms in C++ and CUDA, and conducted experiments on an Intel Xeon 2.90 GHz Linux machine with 128 GB RAM and one NVIDIA GeForce RTX 3090 GPU. We compile our programs with NVCC 11.4 and GNU GCC 10.3. Since KLayout (Köfferlein, 2018) (version 0.26.6) is utilized to complete the end-to-end DRC flow, we use the default DRC functionality in KLayout (8 threads) as the baselines. The designs tested in the experiments are all synthesized from the OpenROAD project (Ajayi et al., 2019).

Runtime Comparisons

We first compare the overall runtime of design rule checks between X-Check and KLayout. The results are shown in Table 6.1 (*width* check) and Table 6.2 (*space* check, *enclosing* check).

Width check is the most simple task among the checks, as it examines violations within each polygon. Although it is less meaningful to discuss speedup when the program already runs fast, we still observe mild performance gain (1.13× and 1.18×) in the two largest cases (i.e., `Metal1` of `bp_be` and `Metal1` of `bp`). Besides, despite the dynamic algorithm selection process, such overhead is negligible (< 0.1s) in all the cases.

For *enclosing* check and *space* check, the CPU version takes a much longer time to complete. Therefore, X-Check achieves a much higher speedup in these cases. For *enclosing* check, GPU-

Table 6.2: Runtime Comparisons of *Enclosing Check* and *Space Check*

Design	Layer	Enclosing Check			Space Check		
		KLayout	X-Check	Speedup	KLayout	X-Check	Speedup
gcd	Metal1	38.4	2.4	16.00×	12.6	2.4	5.25×
	Metal2	2.5	2.5	1.00×	6.4	2.4	2.67×
aes	Metal1	15 470.4	12.3	1257.76×	4493.8	67.5	66.57×
	Metal2	2227.0	14.5	153.59×	2778.5	9.9	280.66×
bp_be	Metal1	66 194.6	128.6	514.73×	6718.7	123.7	54.31×
	Metal2	3089.2	147.4	20.96×	4171.5	16.6	251.30×
bp	Metal1	98 370.4	235.3	418.06×	14 019.7	233.4	60.07×
	Metal2	3958.7	276.6	14.41×	5164.4	65.9	78.37×
Average				61.36×			45.00×

enabled X-Check achieves up to 1257.76× speedup, with an average of 61.36×. For *space* check, X-Check offers up to 280.66× speedup and an average of 45.00× improvement. The significant speedup confirms the effectiveness of our proposed parallel sweepline paradigm.

Runtime Breakdown

We care about the breakdown of runtime because 1) we want to understand where does the speedup come from, and 2) want to foresee where is the new bottleneck for potential further speedup.

Width Check Runtime Breakdown As we have achieved mild speedup for width check, it is desired to profile the application after GPU acceleration for further analysis. Therefore, we collected the runtime statistics of each thread for the largest test case `bp`, with a particular interest in the comparison between the *merge* stage and the *check* stage. The results are shown in Figure 6.5. Each horizontal bar is for one thread, where the purple portion is for the *merge* stage, and the gold portion is for the *check* stage. For KLayout, the *check* stage takes from 21.3% to 56.6% of the runtime, with an average of 39.8%. After GPU acceleration, the *check* stage in X-Check takes from 1.5% to 21.4% of the runtime, with an average of 6.5%. The result matches that in Table 6.1, indicating the source of the current speedup, as well as explaining the limited performance gain.

(a) KLayout (b) X-Check

Figure 6.5: Runtime breakdown of *width check* on `Metal 1` of the `bp` design. The purple and gold portions are for the *merge* and the check stages, respectively.

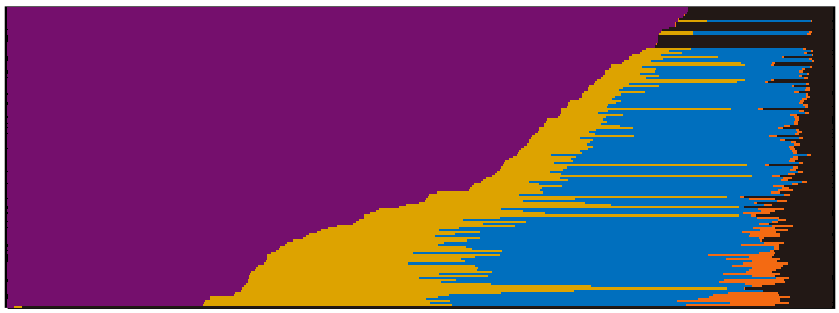


Figure 6.6: Runtime breakdown of *enclosing check* on `Metal 1` of the `bp` design. The purple portion is for *merge*, gold for *sort*, blue for *prefix build*, orange for *violation report*, and **black** for **the rest**, respectively.

Enclosing Check Runtime Breakdown We are also curious about the runtime breakdown of the slow cases. Therefore, we also profiled X-Check on the enclosing check for the `bp` design. The results are shown in Figure 6.6. In the figure, each horizontal bar is for one tile, where the purple portion is for *merge*, gold for *sort*, blue for *prefix build*, orange for *violation report*, and **black** for **the rest**, respectively. Some tiles do not have valid enclosing checks to be performed, so there is no time spent on *sort*, *prefix build*, and *check*. The lowest bar has a substantial portion for 'the rest' because it is the first tile and carries some warm-up jobs for GPU. From the figure, the *merge* stage still takes a significant portion of time (up to 82.5% and averaged 55.9%), indicating the new runtime bottleneck after GPU acceleration of the sweepline algorithm for violation report.

Ablation Study

In this section, we further investigate the effectiveness of some implementation techniques we discussed in Section 6.3.3.

Dynamic Algorithm Selection As introduced, it is not desired to invoke GPU execution if the estimated parallelism is limited. To illustrate the importance of such a strategy, we compare the width check runtime for `Metal 2` of all the designs. For these cases, the average edges per polygon are small - it is unlikely to have performance gain by involving GPU computation. The experimental results, as shown in Figure 6.7, have confirmed the case.

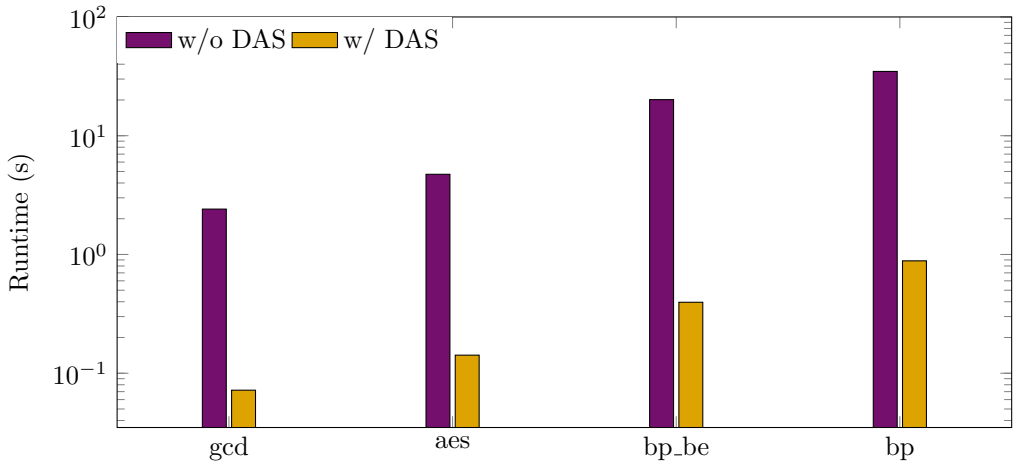


Figure 6.7: Runtime comparisons of width check on `Metal 2`. Runtimes are in log scale. For the sparse tiles, dynamic algorithm selection significantly reduces runtime.

Sorting Strategy Selection Sorting strategies also affect the runtime performance. To demonstrate, we compare the runtime of enclosing check using merely `thrust::sort` (i.e., merge sort), merely Copy-Sort-Permute strategy, and a mixed strategy that switches to CSP when the array size is larger than a predefined threshold (8k in our practice). As shown in Figure 6.8, the mixed strategy indeed outperforms both single strategies.

Besides, we further tested sorting synthetic arrays. In this setting, the array contains `struct`s whose sizes are 48 bytes. The array lengths vary from 2 to at most 2^{25}, and we sort them using both merge sort and the copy-sort-permute (CSP) strategy and compare the performance. The results

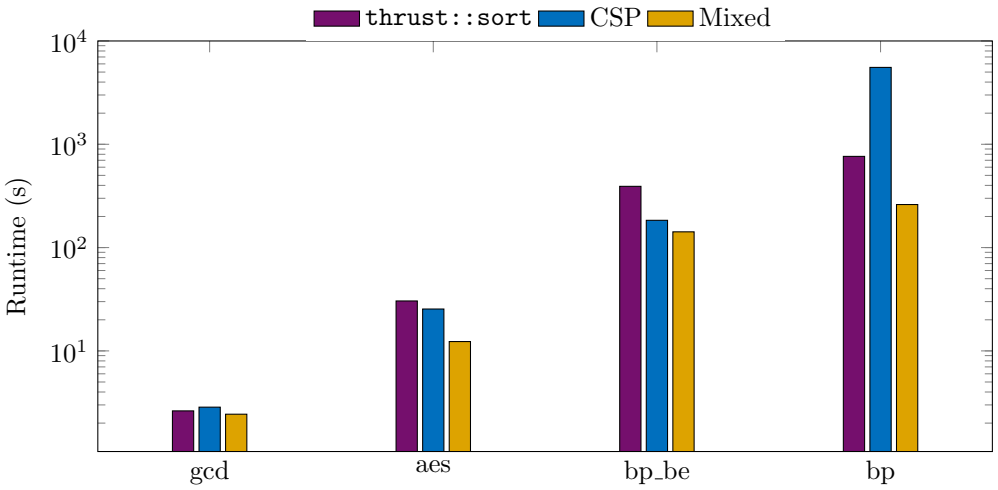


Figure 6.8: Runtime comparisons of enclosing check on `Metal 1` using different sorting strategies. Runtimes are in log scale. The mixed strategy achieves the fastest runtime in all cases.

are shown in Figure 6.9, and note that both axes are in log scale. As can be seen, the CSP strategy runs significantly faster than merge sort for large arrays. CSP outperforms merge sort when the input arrays are large enough. However, CSP runs slower for smaller arrays as the overhead cannot be ignored for those cases. The arrow in the figure points out that CSP wins when the array size is around 65536 or larger.

6.3.5 Summary

Design rule checking is crucial in physical verification. As the size of modern VLSI circuits continues to grow, the demand for parallel, hardware-friendly DRC algorithms have been highlighted. In this section, we have proposed to utilize a parallel sweepline algorithmic paradigm to solve a series of DRC problems. We have analyzed the theoretical complexity of the algorithms, implemented them on GPUs, and further integrated them into an end-to-end DRC flow. We conducted thorough experiments to demonstrate the effectiveness of the algorithms: they have achieved an average of $1.09\times, 61.36\times$, and $45.00\times$ speedup in three different DRC tasks, compared with a multi-threaded CPU design rule checker. We also provided other experimental results for further discussion.

In the future, we would like to investigate parallelizing the *merge* procedure with the sweepline paradigm, as it appears to be the new runtime bottleneck. Besides, we feel it necessary to develop

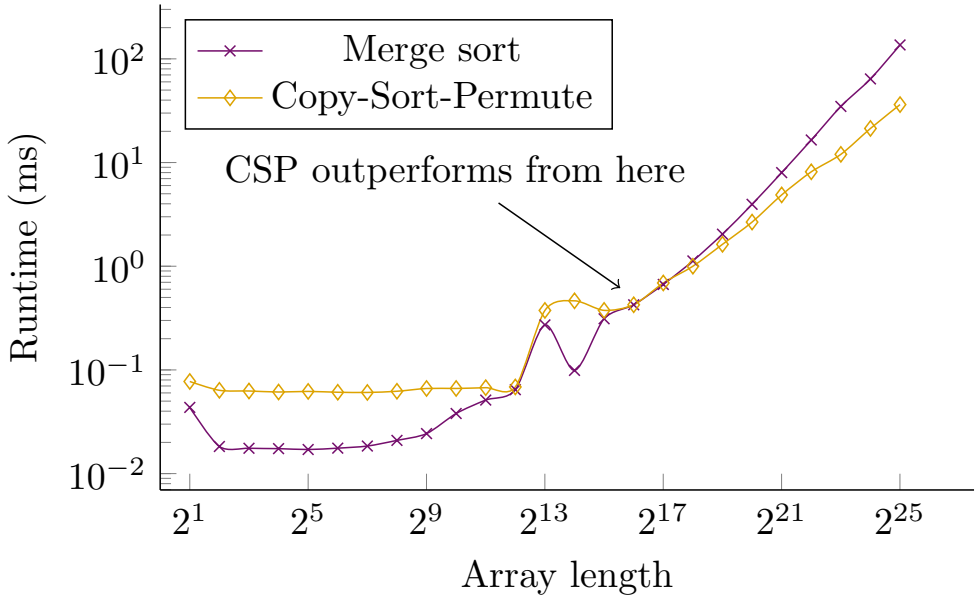


Figure 6.9: Runtime comparisons of merge sort and the CSP sorting strategy. CSP outperforms merge sort when the input arrays are large.

more programming infrastructures for GPUs, including dynamic vectors, associative data structures, and their thread-safe solutions.

6.4 OpenDRC: An Efficient Open-Source Design Rule Checking Engine with Hierarchical GPU Acceleration

6.4.1 Overall Flow

We first introduce the overall flow of OpenDRC, as illustrated in Figure 6.10. Given a hierarchical layout, OpenDRC parses the input file, and maintains the components in a layer-wise bounding volume hierarchy tree (detailed in Section 6.4.2). Meanwhile, design rules are specified from the provided programming interface (introduced in Section 6.4.3). For the layers relevant to the specified design rules, OpenDRC performs an adaptive row-based partition of the layout, which effectively identifies independent regions (detailed in Section 6.4.2). After layout partitioning, OpenDRC provides a sequential (CPU) branch (detailed in Section 6.4.2) and a parallel (GPU) branch (detailed

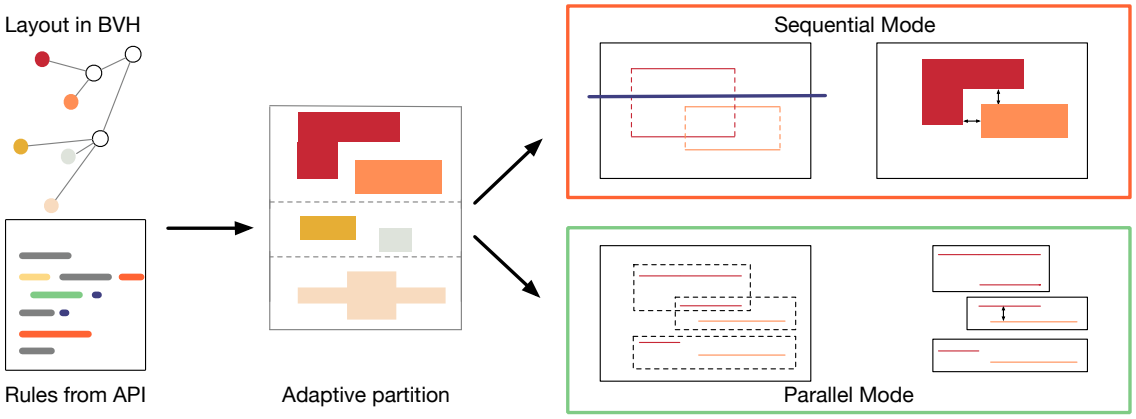


Figure 6.10: The overall flow of OpenDRC.

in Section 6.4.2) to execute the design rule checks.

6.4.2 Algorithms

Layer-wise Bounding Volume Hierarchy

Hierarchical modularity is a natural solution for designers to cope with very large-scale systems. In the `GDSII` stream format (Calma, 1987), infinitely many hierarchical layers could be defined by recursive structure reference:

```
<structure> ::= BGNSTR STRNAME {<element>}* ENDSTR
<element>   ::= { ··· | <SREF> | ···} ··· ENDEL
<SREF>      ::= SREF ··· SNAME ···
```

In the above Backus Naur representation of the stream syntax, a ⟨`structure`⟩ is composed of a list of ⟨`elements`⟩, and an ⟨`element`⟩ could be, among others, a structure reference ⟨`SREF`⟩ that instantiates another structure defined elsewhere. Hereafter we use 'cell' and 'structure' interchangeably. To enable hierarchical design rule checking, OpenDRC does not flatten the layout, but preserves the layout hierarchy instead. Specifically, a structure reference effectively stores a *pointer* to the structure definition to reduce memory consumption.

One drawback of the layout hierarchy is that objects belonging to the same layer could scatter around the hierarchy tree. However, (range) queries for layer objects are very common since

many design rules are defined for specific layers. To improve efficiency for such queries, OpenDRC maintains the *minimum bounding rectangle* (MBR) of each cell; for a cell that spans multiple layers, separated MBRs are computed for each layer and maintained. To answer a layer range query, it suffices to descend the hierarchy tree from the topmost ⟨`structure`⟩ (root), and prunes the whole subtree rooted at an element if its MBR for the interested layer is empty. Augmenting the hierarchy tree with MBRs reduces the layer range query complexity from $O(n)$ to $O(\min(n, kh))$, where n is the number of leaf nodes, k is the number of output, and h is the height of the hierarchy tree. Note that such MBR technique is widely applied in geometric data structures such as kd-trees (Bentley, 1975) and R-trees (Guttman, 1984).

Duplication and inverted indices An effective strategy to trade space consumption for speed is to duplicate the hierarchy tree in a layer-wise manner. Namely, a separated hierarchy tree is built for each layer such that only modules containing objects in that layer are added to this hierarchy tree. The space consumption could be enlarged by at most L times where L is the number of layers. Suppose queries only ask for *all* objects in the given layer, it is possible to further construct element-level inverted indices that each contain a full list of leaf elements belonging to a layer.

Adaptive Row-based Partition

OpenDRC offers an adaptive row-based partition scheme that turns out to be very effective for check pruning and parallelization. The rationale behind is related to the popular row-based placement (Lu et al., 2014; Lin et al., 2019). The intuitions are twofold:

1. Layouts can be partitioned into non-overlapping regions (rows) along the y-axis, where cells do not overlap too much;

2. By grouping cells into independent rows, x-coordinates of cells in a row are more likely to be separated as well.

Technically, the row-based partition can be regarded as an interval merging problem, which can be efficiently solved in $\Theta(k + N)$ time, where k is the number of merge operations, and N is the size of the domain. In our case, k equals the number of cells, and N is the number of unique y-coordinates

(discretization assumed). The algorithm can be divided into three steps: 1) initialize an array A of size N with indices as entry values; 2) merge y-coordinates belonging to the same cell; and 3) scan the whole array A to obtain the cover. To be specific, we use an '*pigeonhole* array' (of domain size N) to maintain the right endpoints of intervals, while interval left endpoints are indicated by the array indices. For each merge, only one array entry is updated in constant time. Algorithm 4 describes the details.

Note that the interval merging problem can also be solved without using the large *pigeonhole array* by sorting the merge targets, which yields an algorithm with time complexity $\Omega(k \log k)$. We come up with our solution since k is typically much larger than N in our problems, and arrays usually have a much better locality.

Algorithm 4 Interval Merging for Adaptive Layout Partition

Require: A set S of intervals to be merged
Ensure: Non-overlapping intervals covering the domain of S

```
Initialize an array A with indices                          ▷ Step1: Initialize
for all interval [l, r] ∈ S do                              ▷ Step2: Merge
    Update A[l] ← max(A[l], r)
end for
Initialize current interval end e ← −1
for the i-th element ∈ A do                                 ▷ Step3: Scan
    if i > e then                                           ▷ moving across interval boundary
        Create a new interval and reset e
    end if
    Update current interval end e ← max(e, A[i])
end for
```

Task Pruning from Hierarchy Tree

With the preserved hierarchy, OpenDRC always attempts to minimize the number of checks that are actually run. Redundancy could occur due to two possible reasons: 1) the check result could be inferred from previously finished checks; and 2) the check could be eliminated because violations *must* or *cannot* happen. In either the case, running an actual check is unnecessary. The former situation commonly occurs in hierarchical layouts, as they usually contain isomorphic modules that preserve geometric invariants under certain transformations, such as reflection and rotation, and under instantiation constructs like ⟨`SREF`⟩ and ⟨`AREF`⟩ in `GDSII` files. The latter situation can also

be improved with the MBR augmented hierarchy tree.

Intra-Polygon Checks As the finest granularity for transformations is usually at the polygon-level, there exist great optimization opportunities for intra-polygon checks. Given an intra-polygon check for a certain layer, OpenDRC performs *depth-first search* (DFS) along the hierarchy tree to locate layer objects. When a specific layer polygon is first encountered, the corresponding check is scheduled to the task graph. If the check is done for a leaf object, a tag is marked to indicate the finished check type. The same tag is marked for a non-leaf module if all submodules and leaf elements belonging to the module have been checked. In this way, OpenDRC tries to reuse check results when visiting a cell reference element: if the corresponding cell has already been checked elsewhere, and the transformations preserve the target properties of the check, the check result could be safely reused.

Inter-Polygon Checks Inter-polygon checks are slightly more complicated as many invariants are no longer preserved under common constructs. Nevertheless, OpenDRC still attempts to explore opportunities to reduce workloads. Given an inter-polygon check between layer M and layer N (M and N could be identical), OpenDRC still searches along the hierarchy tree from the root, denoted as a pair of nodes $(root^M, root^N)$. A similar memoization strategy is used as described in intra-polygon checks. Specifically, only if (a^M, a^N) has been checked, OpenDRC marks it down for possible reuse. Note that the check result of (a^M, b^N) cannot be reused if a and b do not belong to the same parent cell, because another instantiation of them may not be of the same relative position. A check for node pair (a^M, b^N) could possibly be eliminated if:

- $M = N \wedge id_a > id_b$. Node id assignment could be arbitrary. This is a duplication of the check for (b^M, a^N).
- $a = b$ and (a^M, a^N) has been checked. This corresponds to redundancy case 1) we described.
- $\mathrm{MBR}_a^M \cap \mathrm{MBR}_b^N = \emptyset$. This corresponds to redundancy case 2) we described.

Technically, the MBRs should be enlarged by a minimum rule distance to ensure non-overlapping indeed indicates no violations.

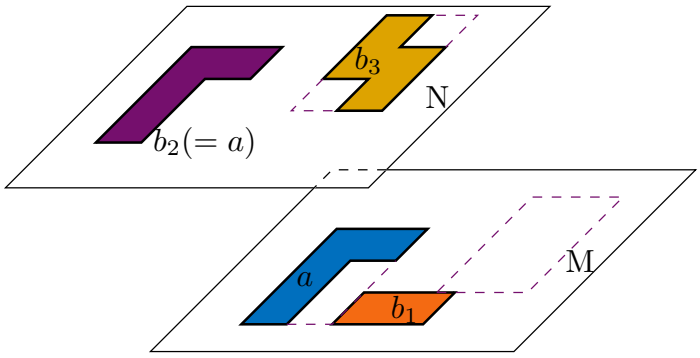


Figure 6.11: Three inter-polygon checks could be eliminated.

Figure 6.11 illustrates the above three cases. When $M = N$, the check (a^M, b_1^N) is a duplication of (b_1^M, a^N). As b_2 and a refer to the same cell, the check result of (a^M, a^N) can be reused if it is already checked. The check (a^M, b_3^N) can be pruned because their MBRs are non-overlapping.

The Sequential Mode

As by the task pruning strategy introduced in Section 6.4.2, the sequential mode of OpenDRC first detects potential violations between objects by querying overlapping MBRs of polygons or cells, and then performs edge-based checks among those object pairs.

Overlapping MBR Query. OpenDRC runs a standard sweepline algorithm (Bentley and Wood, 1980) to detect all overlapping MBRs, except that interval trees (McCreight, 1980) are used instead of segment trees for implementation simplicity. An interval tree is a binary search tree that stores an interval I in the highest node satisfying $u \in I$, where u is the key of this node. Specifically, every node of the interval tree maintains its intervals in two separate lists: one is sorted by left endpoints, and the other is sorted by right endpoints. By the definitions above, all left endpoints (resp. right endpoints) stored in the right (resp. left) subtree are larger (resp. smaller) than the parent node's key, which enables efficient range queries. The sweepline algorithm moves a conceptual line across the plane from top to bottom, which scans through the top and bottom sides of all MBRs in descending y. When the top side of an MBR m is encountered, the corresponding horizontal interval is inserted into the interval tree, and a query to the interval tree reports all the MBRs overlapping with m. When the bottom side of m is encountered, the horizontal interval is removed from the interval tree.

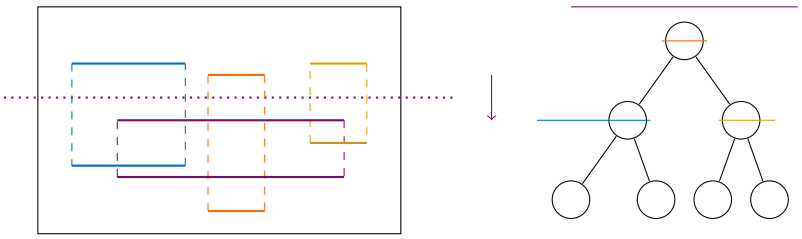


Figure 6.12: Sweepline and interval tree for overlapping MBR query.

Figure 6.12 illustrates the sweepline procedure and the corresponding interval tree.

Check Procedures. For distance rules, edge-to-edge checks need to be performed, be it an intra-polygon check extracted from the hierarchy tree, or an inter-polygon check obtained from MBR queries. Polygon vertices are stored in clockwise order, so that positional relations of edges are determined accordingly. For area rule checks, OpenDRC computes polygon areas by the *Shoelace Theorem*.

The Parallel Mode

The parallel mode of OpenDRC runs design rule checks on GPUs, which utilizes very different algorithms and data structures from those for sequential processing. After layout partitioning, OpenDRC performs parallel design rule checks in a row-by-row manner, as cells belonging to different rows will not produce any violation.

Before checking, OpenDRC packs the edges of relevant polygons into a flattened array, which is transferred from the host memory to the GPU device memory. Depending on the complexity of each polygon or polygon pair, OpenDRC selects either a brute-force executor or a sweepline executor. For smaller tasks, parallel threads are launched for each polygon (or pair), in which edge pairs are enumerated and checked. For larger tasks, a parallel sweepline algorithm is performed, which is similar to the one described in Section 6.3.2: firstly, a parallel *scan* determines the check range of each edge; then parallel threads are launched to perform the check between an edge and all other edges within its check range. Although these two steps can be combined theoretically, separating them into two kernel launches enables efficient kernel code optimization (viz. `for` loops versus `while` loops).

6.4.3 Design and Implementation Details

Software Architecture

Conceptually, OpenDRC consists of four layers, from topmost to bottommost:

1. the `interface` layer,
2. the `application` layer,
3. the `algorithm` layer, and
4. the `infrastructure` layer.

In general, higher layers depend on the abstraction of lower layers, but not the other way around. The `interface` layer is responsible for the interaction between OpenDRC and the outside world, such as reading design files, defining rule decks, adaptors to design databases, and result output. The `application` layer can be regarded as a system controller that schedules computation tasks and dispatches them to algorithms. The `algorithm` layer, as indicated by the name, consists of the implementation of design rule checking algorithms, such as `width-check` and `space-check`. The `infrastructure` layer is for abstract data structure and algorithms, various program utilities (timer, logger, etc.), and some basic GPU libraries.

General Programming Interface

OpenDRC aims to provide extreme extensibility and usability through its general programming interface. OpenDRC recognises the need of researchers and end users to customize their usage of the engine, so it encourages the use of the C++ programming interface, instead of another scripting language, as the default way to define design rule checking tasks. The code snippet in Listing 2 demonstrates how to program OpenDRC. We start by reading-in a layout file and creating an instance of the DRC engine. Then we specify a list of design rules using the `add_rules` method, where each rule is described in *chaining methods* that resemble natural language. In this example, we have defined three rules: the first rule ensures that all the polygons are rectilinear; the second rule ensures the minimal width in layer 19 is 18nm; the third rule ensures that every polygon in layer 20 has a non-empty name. Finally, calling `check()` will run checks for the specified rules.

Listing 2 Code snippet of using OpenDRC.

```
auto db = odrc::gdsii::read(/* path-to-gdsii */);
auto e  = odrc::engine();
e.add_rules({
  db.polygons().is_rectilinear(),
  db.layer(19).width().greater_than(18),
  db.layer(20).polygons().ensures(
      [](const auto& p){return !p.name.empty();}
  )
});
e.check(db);
```

OpenDRC defines two categories of methods, *selectors* and *predicates* to help define design rules. Selectors locate the target objects for which a design rule is defined. In our example, chained methods `layer(19).width()` selects the width in the 19-th layer as the check target. Predicates are the conditions that the selected objects need to conform to, such as `is_rectilinear()` that requires axis-aligned shapes. Specifically, the `ensures()` method takes a callable as a parameter that enables user-defined predicates.

Heterogeneous Computing via Asynchronous Operations

A CPU-GPU computing platform is heterogeneous, which requires special considerations on the orchestration between them. OpenDRC utilizes asynchronous operations and Stream Ordered Memory Allocator (NVIDIA, 2023) to hide communication or computation latencies. When OpenDRC starts, it creates CUDA stream objects that are responsible for asynchronous operations. As the parsing is finished and the database is ready, asynchronous data copies are launched to prepare necessary data (e.g., polygon edges) for parallel checks. The data movement is thus usually hidden by the layout partitioning in the flow. OpenDRC also tries to overlap CPU computation and GPU processing to hide latency. For example, parallel checks of a row (taking place on device) can be performed concurrently with the necessary data preprocessing of the next row (taking place on host).

Functors and Type Traits

The extensibility of OpenDRC also comes from a generic implementation of underlying functors. The sweepline functor shown in Listing 3 is a typical example, which is regarded as a metafunction

that takes another callable as a parameter. Here an executor is either a wrapper for a CUDA

Listing 3 The sweepline functor.

```
1 template <typename Executor, typename EventIt,
2           typename Status,   typename Op>
3 void sweepline(Executor&& exec, EventIt first,
4                EventIt last, Status* st, Op op) {
5   if constexpr (std::is_same_v<std::remove_cv_t<
6     std::remove_reference_t<decltype(exec)>>,
7     odrc::execution::sequenced_policy>) { // CPU
8   } else { /* GPU */ }
9 }
```

stream object (`cudaStream_t`), indicating the operation will be appended to the stream, or a simple `odrc::sequenced_policy` object that indicates a sequential operation.

OpenDRC utilizes type traits to manage certain properties of rules and checks, which dispatches function calls at compile time and avoids runtime branching. Line 5-8 in Listing 3 demonstrates how OpenDRC decides whether a sweepline operates on CPU or GPU by accessing the type traits of the executor in a *constexpr if statement*. Another typical usage is to mark the rule types by the target edge relations (e.g., width or space), which is also implemented in KLayout (Köfferlein, 2018) as runtime arguments. In general, using type traits slightly improves runtime efficiency and helps organize code logic concisely.

6.4.4 Experimental Evaluation

OpenDRC is implemented in C++17 and CUDA. Experiments were conducted on a Linux machine with an Intel Core i7-11700 processor (2.5GHz), 64GB main memory, and an NVIDIA GeForce GTX 1660Ti graphics card. Benchmark layouts are synthesized from OpenROAD (Ajayi et al., 2019), with the ASAP7 (Clark et al., 2016) process design kit (PDK) and all default settings provided in the flow scripts.

To evaluate the efficiency of OpenDRC, we compare its performance with the state-of-the-art multi-threading design rule checker, *KLayout* (Köfferlein, 2018), and the state-of-the-art GPU design rule checker, *X-Check* (He et al., 2022). KLayout provides three different operation modes, namely *flat* mode, *deep* (hierarchy) mode, and *tiling* mode. In the deep mode, the operations will be performed in a hierarchical fashion; in the tiling mode, operations are evaluated in tiles, and multi-

Table 6.3: Runtime comparisons for intra-polygon design rule checks.

Design	Rule	KLayout			X-Check	OpenDRC		Rule	KLayout			X-Check	OpenDRC	
		flat	deep	tile		Seq.	Par.		flat	deep	tile		Seq.	Par.
aes	M1.W.1	3.45	12.69	0.49	0.41	0.02	0.03	M1.A.1	3.34	3.32	0.65	-	0.02	0.03
	M2.W.1	1.37	3.83	0.23	0.14	0.04	0.04	M2.A.1	1.35	1.33	0.37	-	0.04	0.04
	M3.W.1	2.52	2.98	0.36	0.11	0.03	0.03	M3.A.1	2.49	2.51	0.51	-	0.03	0.03
ethmac	M1.W.1	11.88	45.84	1.56	1.21	0.07	0.08	M1.A.1	11.55	11.55	2.05	-	0.07	0.08
	M2.W.1	3.76	10.72	0.52	0.42	0.10	0.11	M2.A.1	3.62	3.63	1.01	-	0.10	0.11
	M3.W.1	6.36	7.64	0.77	0.31	0.08	0.08	M3.A.1	6.20	6.24	1.24	-	0.08	0.08
gcd	M1.W.1	0.13	0.44	0.13	0.11	< 0.01	< 0.01	M1.A.1	0.13	0.13	0.13	-	< 0.01	< 0.01
	M2.W.1	0.05	0.08	0.05	< 0.01	< 0.01	< 0.01	M2.A.1	0.05	0.05	0.05	-	< 0.01	< 0.01
	M3.W.1	0.06	0.07	0.06	< 0.01	< 0.01	< 0.01	M3.A.1	0.06	0.06	0.06	-	< 0.01	< 0.01
ibex	M1.W.1	3.60	12.38	0.50	0.43	0.02	0.03	M1.A.1	3.52	3.52	0.65	-	0.02	0.03
	M2.W.1	1.30	3.61	0.24	0.14	0.03	0.04	M2.A.1	1.27	1.28	0.36	-	0.04	0.04
	M3.W.1	2.38	2.88	0.36	0.10	0.03	0.03	M3.A.1	2.36	2.35	0.51	-	0.03	0.03
jpeg	M1.W.1	13.32	55.35	1.68	1.39	0.08	0.08	M1.A.1	13.01	13.00	2.17	-	0.07	0.08
	M2.W.1	3.05	8.77	0.46	0.40	0.10	0.10	M2.A.1	2.98	2.95	0.95	-	0.09	0.09
	M3.W.1	4.86	6.14	0.59	0.29	0.08	0.08	M3.A.1	4.79	4.81	1.10	-	0.08	0.07
sha3	M1.W.1	3.48	12.36	0.49	0.43	0.02	0.03	M1.A.1	3.40	3.40	0.63	-	0.02	0.03
	M2.W.1	1.10	2.95	0.21	0.12	0.03	0.03	M2.A.1	1.07	1.09	0.33	-	0.03	0.03
	M3.W.1	1.79	2.15	0.30	0.09	0.02	0.02	M3.A.1	1.79	1.77	0.42	-	0.02	0.02
uart	M1.W.1	0.15	0.40	0.15	0.11	< 0.01	< 0.01	M1.A.1	0.14	0.14	0.15	-	< 0.01	< 0.01
	M2.W.1	0.06	0.12	0.06	< 0.01	< 0.01	< 0.01	M2.A.1	0.06	0.06	0.06	-	< 0.01	< 0.01
	M3.W.1	0.08	0.09	0.08	< 0.01	< 0.01	< 0.01	M3.A.1	0.08	0.08	0.08	-	< 0.01	< 0.01
Average		37.7×	82.1×	9.6×	4.5×	0.9×	1.0×		37.6×	37.6×	13.0×	-	1.0×	1.0×

CPU support is enabled (Köfferlein, 2018). These three modes are exclusive, so we list DRC runtime under the three options in individual columns since no combination of them is directly accessible. We reimplement the vertical sweeping algorithm proposed in X-Check (Section 4.1 in their paper (He et al., 2022)).

We follow the experimental settings in X-Check (He et al., 2022) to check (minimum) *width*, *spacing*, and *enclosure* rules; we further implement *minimum area* checks that X-Check is unable to deal with. These rules are typical, as they include two intra-polygon rules (*width*, *area*) and two inter-polygon rules (*spacing*, *enclosure*); *enclosure* rules are inter-layer while others are intra-layer; except *area* rules, other rules are essentially *distance* rules. The selected rules involve Back-End-Of-Line (BEOL) layers M1, M2, M3, V1, and V2 (Clark et al., 2016).

Runtime comparisons for intra-polygon checks are shown in Table 6.3, and comparisons for inter-polygon checks are in Table 6.4.

The 'average' rows are normalized against the parallel mode of OpenDRC, where the runtime

Table 6.4: Runtime comparisons for inter-polygon design rule checks.

Design	Rule	KLayout			X-Check	OpenDRC		Rule	KLayout			X-Check	OpenDRC	
		flat	deep	tile		Seq.	Par.		flat	deep	tile		Seq.	Par.
aes	M1.S.1	4.33	13.78	0.62	0.17	0.21	0.06	V1.M1.EN.1	468.24	462.28	15.97	0.20	6.44	0.12
	M2.S.1	1.55	4.15	0.29	0.13	0.09	0.02	V2.M2.EN.1	2.93	1.64	0.59	0.14	0.18	0.09
	M3.S.1	2.64	3.25	0.38	0.12	0.15	0.02	V1.M2.EN.2	469.96	468.89	15.71	0.20	0.24	0.12
ethmac	M1.S.1	14.67	48.50	1.89	0.39	0.72	0.14	V1.M1.EN.1	3045.02	3038.10	57.76	2.00	42.35	0.41
	M2.S.1	4.35	11.71	0.59	0.20	0.23	0.05	V2.M2.EN.1	8.29	4.74	1.45	0.23	0.47	0.22
	M3.S.1	6.68	8.17	0.82	0.16	0.39	0.04	V1.M2.EN.2	3031.20	3034.67	55.63	0.36	0.84	0.32
gcd	M1.S.1	0.15	0.46	0.14	0.11	< 0.01	0.01	V1.M1.EN.1	3.06	2.96	3.09	0.11	0.06	< 0.01
	M2.S.1	0.05	0.09	0.05	0.11	< 0.01	< 0.01	V2.M2.EN.1	0.07	0.05	0.08	0.10	< 0.01	< 0.01
	M3.S.1	0.06	0.07	0.06	0.11	< 0.01	< 0.01	V1.M2.EN.2	2.95	2.95	2.99	0.10	< 0.01	< 0.01
ibex	M1.S.1	4.45	13.15	0.63	0.17	0.22	0.06	V1.M1.EN.1	477.86	473.62	16.03	0.21	7.14	0.13
	M2.S.1	1.49	3.96	0.25	0.13	0.09	0.02	V2.M2.EN.1	2.78	1.56	0.56	0.15	0.18	0.08
	M3.S.1	2.50	3.08	0.39	0.12	0.14	0.02	V1.M2.EN.2	479.79	477.17	15.90	0.17	0.24	0.12
jpeg	M1.S.1	15.82	57.36	2.01	0.43	0.80	0.16	V1.M1.EN.1	3609.55	3580.46	58.29	1.59	55.07	0.49
	M2.S.1	3.48	9.79	0.49	0.21	0.20	0.05	V2.M2.EN.1	7.07	4.04	1.22	0.22	0.40	0.20
	M3.S.1	5.17	6.70	0.64	0.16	0.30	0.03	V1.M2.EN.2	3611.69	3588.04	57.01	0.35	0.87	0.32
sha3	M1.S.1	4.23	13.02	0.60	0.16	0.21	0.06	V1.M1.EN.1	476.10	472.44	15.87	0.49	7.07	0.12
	M2.S.1	1.16	3.23	0.22	0.12	0.07	0.02	V2.M2.EN.1	2.32	1.29	0.48	0.13	0.14	0.07
	M3.S.1	1.87	2.31	0.30	0.11	0.11	0.02	V1.M2.EN.2	468.70	467.92	17.28	0.15	0.22	0.11
uart	M1.S.1	0.19	0.44	0.19	0.11	< 0.01	0.01	V1.M1.EN.1	3.61	3.50	3.62	0.10	0.06	< 0.01
	M2.S.1	0.07	0.13	0.07	0.11	< 0.01	< 0.01	V2.M2.EN.1	0.10	0.06	0.10	0.12	< 0.01	< 0.01
	M3.S.1	0.08	0.10	0.08	0.10	< 0.01	< 0.01	V1.M2.EN.2	3.49	3.48	3.54	0.10	< 0.01	< 0.01
Average		47.6×	99.5×	12.0×	5.6×	3.2×	1.0×		514.9×	429.0×	61.5×	2.9×	4.7×	1.0×

is the *geometric mean* of the column, as we value all checks equally regardless of their sizes. Note again that X-Check is unable to perform area checks, so the column is empty. Intra-polygon checks generally run fast, which confirms the claim in X-Check (He et al., 2022). OpenDRC achieves 4.5× speedup on average compared with X-Check, and 9.6× - 13.0× speedup compared with KLayout (tiling mode). For sequential modes, OpenDRC is around 37.6× faster than the flat/deep mode of KLayout, which we argue is due to the hierarchy strategy OpenDRC adopts. In fact, both sequential and parallel modes of OpenDRC run equally fast for intra-polygon checks. Inter-polygon checks have heavier computation workloads, where we see more significant speedup from GPU acceleration. For space checks, GPU-accelerated OpenDRC is 3.2×, 5.6×, and 12.0× faster than sequential OpenDRC, X-Check, and KLayout (tiling mode), respectively; for enclosing checks, the speedups become 4.7×, 2.9×, 61.5×, respectively.[4] The sequential implementation of OpenDRC is also 14.9× - 91.3× faster than KLayout (the faster one in flat/deep mode). The experiments demonstrate the efficiency of

[4]We notice the abnormal runtime of KLayout reported in X-Check (He et al., 2022), which we think could be due to a very large number of violations that trigger abnormal program behavior (e.g., heavy disk IO, etc.).

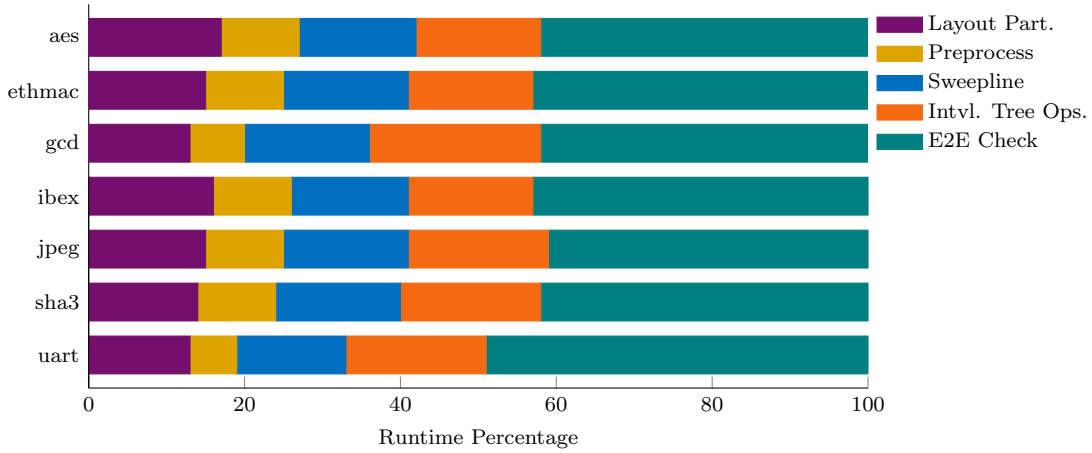


Figure 6.13: The runtime breakdown of OpenDRC sequential *minimum spacing* checks. 'Layout Part' refers to adaptive layout partitioning; 'Intvl. Tree Ops.' refers to interval tree operations `insert`, `remove`, and `query`; 'E2E Check' refers to edge-to-edge checks.

OpenDRC and the effectiveness of the proposed techniques.

We also provide a runtime breakdown of OpenDRC in Figure 6.13, taking sequential space checks as an example. Since asynchronous operations are utilized in the parallel mode, runtime profiling and visualization are slightly complicated and are left to future work. As can be seen, the adaptive layout partition consumes only around 15% of overall runtime, but greatly enhances the efficiency of subsequent steps. The sweepline algorithm, together with operations in the interval tree, taking around 35% of runtime, examines overlapping of cell MBRs and prunes unnecessary checks. Finally, 40% - 50% of the overall runtime is spent on edge-to-edge space checks.

6.4.5 Summary and Roadmap

As inspired by many interesting research problems in VLSI layout operations and design rule checking, we develop OpenDRC, a new open-source design rule checking engine. By introducing adaptive row-based layout partition and efficient sequential/parallel hierarchical DRC procedures, OpenDRC achieves significant speedup compared with state-of-the-art multi-threading and GPU design rule checkers. Ongoing works for OpenDRC include a systematic evaluation of heterogeneous computing in DRC, data compression techniques for memory footprint reduction, and supports for general geometric shapes.

6.5 Discussion: Developing An STL-like Parallel Programming Library for GPU

6.5.1 Introduction

GPUs are ubiquitous nowadays as they offer massive parallel computing power to support modern applications like high performance computing and deep neural network execution. While parallel computing is a common solution due to its high peak performance, programming parallel devices is nontrivial, and the programmer needs to understand hardware details to obtain good software performance.

Domain-specific languages offer another direction to ease GPU programming. Many functional data-parallel languages (e.g., *Accelerate* (McDonell et al., 2013) and *Futhark* (Henriksen et al., 2017)) are invented, in which parallelism are implicitly expressed by higher-level parallel constructs, such as `map`, `reduce`, and `scan`. The compiler is responsible for the implementation details of the constructs, which highly affects the performance of the generated program. In this way, the programming efforts are greatly reduced since the compiler abstracts the programmer away from GPU architecture details. Still, the programmers do not have very fine-grained control over the programs. In other words, the generated program is a black box to the user, which prevents the analysis or reasoning of the program. Another design option is to enable lower-level parallel programming by using extra directives (e.g., `#pragma` in `C++`), such as *PGI* (Wolfe, 2010). These directives are used to annotate the original program to guide compiler decisions, such as how large the loop tiling size is or which memory to allocate.

Besides inventing a new programming language, developing a GPU programming library is an easier choice with probably lower starting efforts. The domain-specific libraries are commonly seen, each dedicated to a particular domain, such as deep learning and graph processing. *CuDNN* (Chetlur et al., 2014) provides an efficient implementation of deep learning primitives such that the programmers can achieve high efficiency on deep learning operations without having a deep understanding of parallel architecture. Similarly, the *cuSPARSE* (Naumov et al., 2010) library provides a set of basic linear algebra subroutines used for handling the computation of sparse matrices. In graph pro-

cessing, *Gunrock* (Wang et al., 2016) is a library that delivers high performance in computing graph analytics and allows programmers to develop new graph primitives with minimal GPU programming knowledge quickly. Apart from domain-specific libraries, there are also GPU programming libraries that are designed to be more generic. *ArrayFire* (Malcolm et al., 2012) and *Boost.Compute* (Szuppe, 2016) provide massive capability for GPU computing on CUDA and OpenCL capable devices, in which a rich set of fine-tuned functions are provided for various domains including linear algebra, convolutions, etc. Moreover, STL-like GPU programming libraries offer several generic GPU data structures for implementing high-performance parallel applications with minimal programming effort. *Thrust* (Bell and Hoberock, 2012) is a C++ template library for CUDA based on the STL, which provides an extensive collection of data-parallel primitives such as scan, sort, and reduce. *Stdgpu* (Stotko, 2019) provides several GPU data structures as the counterparts to the containers defined in the C++ STL, as well as a large set of management functionality and guarantees to improve productivity.

In this section, we follow the philosophy of *Thrust*, to propose *IcyVeins*, an STL-like parallel programming library for GPU. We mainly demonstrate how interface design benefits standard programming techniques like operator fusion (Section 6.5.3) and data layout optimization (Section 6.5.3). To adapt library performance to various computing platforms, we also carry out comprehensive experiments in Section 6.5.4 to analyze the choice of execution configurations.

6.5.2 Preliminary

Standard Template Library

Templates in the C++ programming language are a typical abstraction for generic programming, where data types are passed/specified as parameters to avoid code duplication for different data types. Templates are expanded at compile-time (i.e., with low runtime overhead) with concrete data type arguments. Therefore, one template may be compiled to multiple instances of the same logic with varied types.

Specifically, the Standard Template Library (STL) is a generalized software library that provides a series of parameterized *containers*, *algorithms*, and *iterators* to boost productivity. *Containers*

store objects and data, and typical containers include vector, list, and map. *Iterators*, implemented by some pointer-like objects, are dedicated to accessing and traversing through a range of elements of *containers*. *Algorithms* define a collection of functions, which perform on the content of *containers* for various targets like sorting, searching, and reversing. The *algorithms* in STL are decoupled from *containers*, which significantly reduces the complexity of the library.

6.5.3 Programming Interface

Library Interface

The library interface basically follows the C++ STL interface, with minor adaptation to parallel programming semantics. Similar to the abstraction in C++ STL, our library consists of three main generic components, namely *containers*, *iterators*, and *algorithms*.

Container A container manages data storage on the device side. It provides member functions to access the elements directly or through iterators.

Iterator Iterators are pointer-like objects that provide a standard interface to step through elements of a container. Typically, two iterators (*begin* and *end*) represent a range of data.

Algorithm Algorithms define functions for various purposes (e.g., searching, counting, manipulating) on a range of elements.

This paper presents a minimal collection of components to help illustrate several exciting ideas behind the library's design philosophy and implementation techniques. To be specific, we use the `vector` container, and two commonly used algorithmic patterns `reduce` and `transform`, as well as their lazy counterparts `reduce_view` and `transform_view` (introduced in Section 6.5.3).

- `icv::vector`
- `icv::reduce`
- `icv::reduce_view`
- `icv::transform`

- `icv::transform_view`

Listing 4 gives an example of computing the squared Euclidean norm of a vector using the library.

Listing 4 Squared Euclidean norm of a vector with IcyVeins.

```
#include <icyveins/algorithm.h>
#include <icyveins/vector.h>
int main() {
    icv::vector<int> vi(100, 0);
    auto view = vi
            | icv::transform_view(icv::square)
            | icv::reduce_view(icv::plus);
    return 0;
}
```

Operator Fusion through Lazy Evaluation

Operator fusion potentially eliminates memory access for intermediate results, which may result in better performance. Consider the two (pseudo) functions in Listing 5 that applies two transformation f and g onto elements in the container v: in `task_v1`, the elements in v are loaded and stored twice

Listing 5 A motivating example of operator fusion.

```
// V is some container type
void task_v1(V &v) {
    transform(v, f);
    transform(v, g);
}
void task_v2(V &v) {
    // gof is the composition of f and g
    transform(v, gof);
}
```

since there are two separated transformation calls. In `task_v2`, however, elements in v are exactly loaded once, transformed by gof (the composition of f and g), and then stored back exactly once. Therefore, it is commonly recommended in many best practice guides to fuse many operators in one heavy function, which usually harms the program's readability.

What is the actual difference between `task_v1` and `task_v2`? In `task_v1`, the operations for each entry are

$$\texttt{LOAD} \to f \to \texttt{STORE} \to \texttt{LOAD} \to g \to \texttt{STORE}.$$

In `task_v2`, the operations for each entry are

$$\texttt{LOAD} \rightarrow f \rightarrow g \rightarrow \texttt{STORE}.$$

We can see that the only difference lies in the middle between f and g, where the intermediate results are immediately stored back and then read out again in `task_v1`. Given that, we attempt to remove such overhead implicitly, without significant overhead in terms of both readability and performance.

Lazy Evaluation serves as a good technical way to abstract operator fusion. With a lazy semantic, operations are executed only when the result is finally required; otherwise, they are lightweight objects that store a promise to execute.

Listing 6 demonstrates the interface of `icyveins::views:: transform`, which is the lazy version of `icyveins::transform`. In our design, the template class `icyveins::views::transform` takes three template parameters, namely a functor type `Callable`, a closure type `ClosureType`, and an iterator type `InputIt`. A transform operator view can be constructed using a single functor (e.g., a lambda expression), leaving the closure and the iterators empty defaults (Line 12).

Operators can be chained together using a pipe operator `|` (Line 24-29). Besides, a container can be bound to an operator chain (Line 17-23) so that the operators are to be executed on the container. We use the `type_traits` standard library to differentiate the above two cases: an instance of the `transform` operator is always derived from the dummy base class `view`, while containers are not. Therefore, we use `std::is_base_of` to examine the inheritance relationship, and use `std::enable_if` to conditionally select the relevant function overloads. If the left hand side operand of the pipe is a container, then the `_begin` and `_end` iterators are updated by calling the `begin()` and `end()` methods of the container. If both operands are operators, a new `transform` object will be constructed, where the left-hand side operator and its own closure are captured in the`_closure` context (because they will execute first), while the right-hand side operator is stored in the `_func` member variable.

When the result of the view is finally required, the operators will be invoked one by one from left to right. This is implemented using constexpr if statement (`if constexpr`): we use `std::is_same<view, ClosureType>` to check if the closure of a transform is a default empty, meaning it is the leftmost operator that should be invoked directly; otherwise, it recursively invokes the operators of

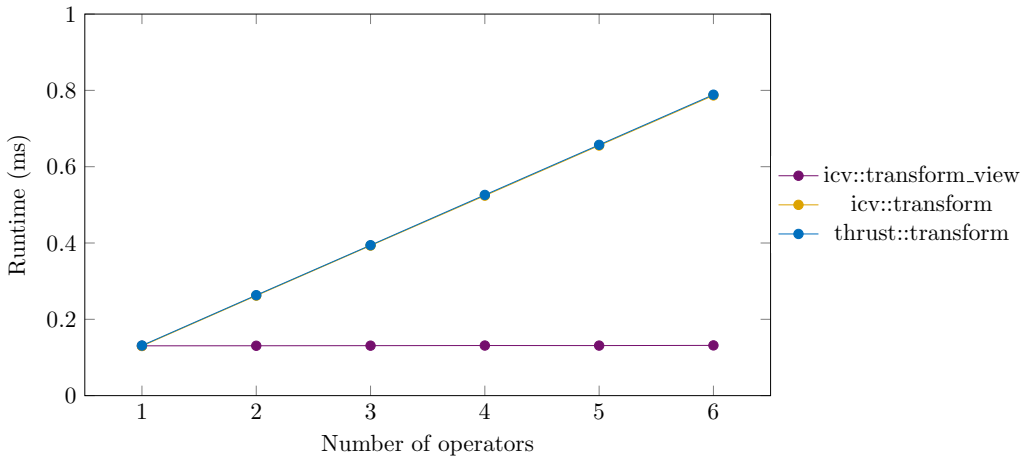


Figure 6.14: Experimental results for operator fusion.

its closure and operates on top of that result. Therefore, the call chain (or the piped views) can be imagined as a linked list of operators (or of single views). In this way, within one `LOAD` and one `STORE`, multiple operations can be performed, and no intermediate main memory access is needed. To confirm that, we can compile some *CUDA* code into *PTX* to check the assembly behaviours. Listing 7 shows such an example, where three transform views are chained together using the pipe operator. From the resulting *PTX* assembly, we can see there are three consecutive `add` instructions between a `ld` introcution and a `st` instruction, which meets our expectation.

Enabling the `reduce` pattern We implemented another computational pattern `reduce` with the same design philosophy as `transform`. The pattern requires a user-defined binary operator (e.g., addition, multiplication, etc.), passed as a parameter to `reduce`, to accumulate multiple entries in a *container* into a single object. By combining `reduce` and `transform`, we can concisely implement complex functions (e.g., Squared Euclidean norm) and apply the STL-like parallel programming library in more applications.

To keep consistency in the interface, we use a similar method in `transform` to overload operator `|`, chaining all operations with the same pipe operator. Considering that the left operand of `|` can be either a `vector`, a `transform`, or another `reduce`, our library handles the different situations separately. If the left-hand side operators are `transform`s, they will be invoked in order as introduced

above.

Data Layout Optimization

Data layout optimization is a commonly used technique to optimize memory access patterns. Consider the example in Listing 8, where we want to store a vector of tuples of an integer (`int`) and a double-precision floating number (`double`). We have two options to explicitly store the data in an *array of struct* (AoS) manner or in a *struct of array* (SoA) manner. The AoS manner (Line 2-5) is more intuitive from a programmer's perspective, since each element in the vector is exactly a tuple (`icv::tuple<int, double>`); when we are going to access one field of the tuples, we directly loop over the vector, obtain each tuple, and access the corresponding field in the tuple. The SoA manner (Line 8-12), however, explicitly splits fields of the tuple into different vectors (`icv::tuple<icv::vector<>...>`). When we are going to access a single field of the original tuples, we simply access the corresponding field in the vector tuple.

Our goal is to provide a simple interface for users to switch between the two data layouts. One of the best possible designs perhaps is to add an extra template parameter to control the layout, as shown in Listing 9. In this way, the user only needs to specify the target layout (`Layout::AoS` or `Layout::SoA`), and then uses the vector in a possibly natural way. Therefore, the remaining problem lies in the implementation. The AoS implementation should be relatively trivial, since we can wrap a normal vector, and forward the functions to the underlying vector class. We now discuss how to implement the SoA version. There are basically two tricky points: 1) how to dynamically dispatch the values into corresponding vectors, and 2) how to correctly arrange host/device memory access.

Dynamic dispatching is solved by using template parameter pack, which is a template parameter that accepts zero or more template arguments. A parameter pack can be expanded to a comma-separated list of zero or more patterns. Specifically, we use `std::integer_sequence` to represent a compile-time sequence of integers that counts the fields in a tuple. As elaborated in Listing 10, we use a private helper function `_get` to deal with the reloaded `operator []`. Here, `_get` accepts a single template parameter `Ids`, which is an integer sequence that will be expanded in `icv::get<Ids>(_sv)[index]` to obtain elements in various fields indexed by `index`. These elements are used to construct a target value of type `tuple<T &...>`. When calling this helper function, we

use `std::make_integer_sequence` to generate such integer sequence according to the size of the parameter pack (i.e., number of fields of the tuple), namely a sequence of $\{0, 1, \cdots, N-1\}$ where N equals to the size of the parameter pack.

It is crucial to keep in mind that the host and the device have separated execution spaces and memory spaces. In our design philosophy, the library interface consists of host functions to ease GPU programming, while the data are stored in device memory to enable parallel manipulation. This inherent mismatch between execution spaces and memory spaces brings about two obstacles: 1) host functions cannot access device memory and vice versa, and 2) `__device__` functions are not allowed to call `__host__` functions and vice versa, unless the callee is marked as `constexpr` and the experimented flag `--expt-relaxed-constexpr` is enabled. Looking back to Line 11 in Listing 9, if the `layout_vector` object is instantiated in the host memory, the `icv::tuple<>` member variable is also a host variable. However, the pointers stored in the tuple must point to device memory locations. Therefore, there is no direct way to access a single entry from the host side (and we should not be able to!). On the other hand, if we are going to access the device data from a `__device__` or `__global__` function, we have to copy the class object or the tuple variable to the device memory, which should be light-weight enough because only the array pointers instead of data are copied. Listing 11 demonstrates a simple use case to switch from SoA to AoS. Note that the parameters of the `__global__` function are pass-by-value ones, where implicit copies are made.

Iterator of SoA To cooperate with algorithms, the SoA container should be equipped with iterators as well. Since the container supports random access, its iterator should be tagged with the `random_access_iterator_tag`. We implement the iterator according to the specification, where the key idea is to store an index in the iterator and access the fields using the index.

Result We compare the performance of different layouts. We create two vectors of `tuple<int, double>` using the two layouts, and test the runtime needed to obtain the `int` field with a simple transformation function. The experiments were carried out on a GeForce RTX 2080 Ti Graphics Card, with vector length 2^{23}, and the operation is repeated for 1000 times to minimize random errors. The results are shown in Table 6.5, where we can see SoA significantly outperforms AoS, as we expect. Even the performance of the AoS vector is comparable to the Thrust baseline (Bell and

Table 6.5: Experimental results for layout optimization.

Vector	AoS	SoA	Thrust (Bell and Hoberock, 2012)
Runtime (ms)	308.154	**131.129**	308.431

Hoberock, 2012) with nearly the same runtime.

Extension to `structs` In our demonstration, we use `tuples` to elaborate the design considerations and implementation details of the data layout switch. Since C++ is not officially equipped with reflection (e.g., `reflexpr` (ISO/IEC TS 23619:2021, 2021)), it is nontrivial to extend the above methodology to deal with structs, for which we do not have universal helper functions like `get`. Nevertheless, this is still somehow achievable using some "Substitution Failure Is Not An Error" (SFINAE) solution, such as matching a `struct` against an initialization list `{std::any...}` (of various lengths) to determine the number of its fields. The tools provided in `boost::pfr` is a good starting point, and interested readers may refer to it for further information.

6.5.4 Performance Adaptability

Execution configurations in CUDA define the dimension of grids and blocks used to execute a function on the device, which in general specifies resource usage to strike for a good performance. Some of the recent works propose to use design space exploration (DSE) methods to search for an exemplary configuration for a specific application, typically DNN deployment (Chen et al., 2018; Sun et al., 2021; Zhao et al., 2021). However, the library could not know the target application beforehand, so it is unlikely to perform DSE at the design time. Besides, it is difficult to accurately model/predict performance, one reason of which, as pointed out by Volkov (Volkov, 2010), is that lower occupancy (i.e., lower thread-level parallelism) may lead to higher instruction-level parallelism. Therefore, we decided to carry out experiments to see how is the overall performance related to different execution configurations.

The performance is highly related to 1) the platform, 2) the problem size, and 3) the task complexity. We come up with the following settings: 1) platforms include a server with a GeForce RTX 2080 Ti graphics card, a server with a GeForce RTX 3090 graphics card, and a Jetson Xavier

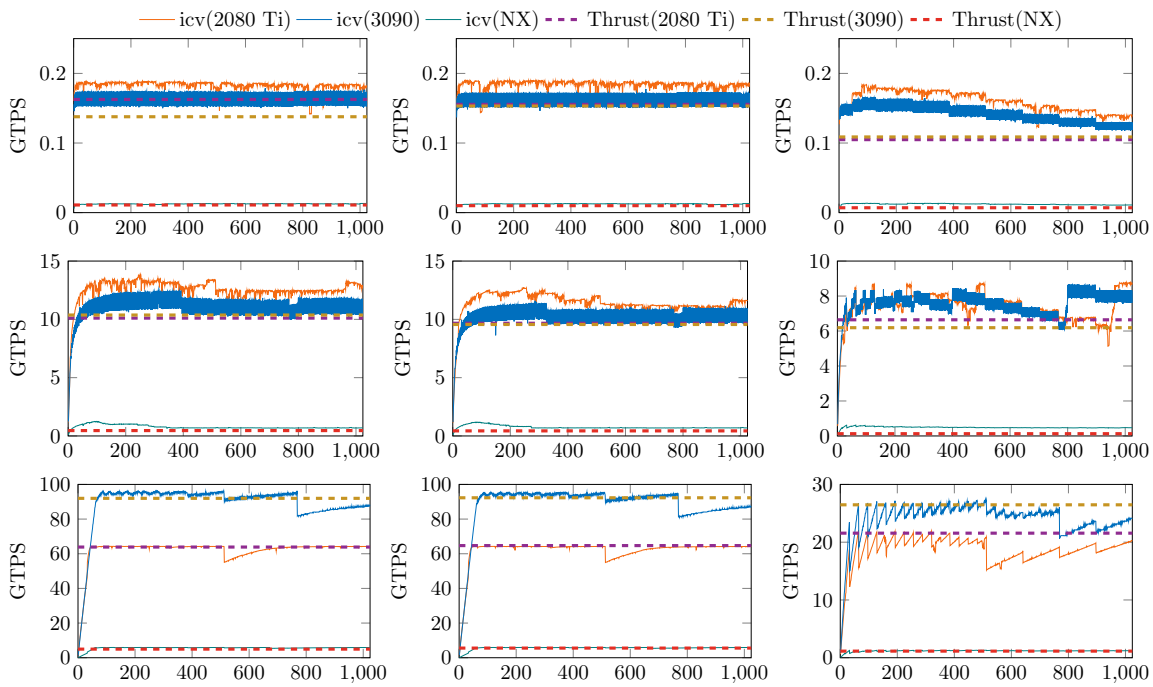


Figure 6.15: Experimental results for execution configurations. The three rows from top to bottom are for vectors of size 1k, 2^{16}, and 2^{23}. The three columns from left to right are for tasks *add*, *polynomial*, and *loop*.

NX embedded device; 2) sizes include 1000, 2^{16}, and 2^{23}; and 3) tasks include a simple addition, a polynomial calculation of degree 5, and a loop of 100 numerical operations ($3a + 1$ for the even iterations and $(a - 1)/3$ for the odd iterations). We then run grid experiments, namely all the possible combinations of the above settings, and the results are illustrated in Figure 6.15. In each sub-figure, the x-axis stands for the number of threads per block, and the y-axis denotes the system throughput. The colored curves denote the results, and the horizontal dashed lines are the baseline performance of the Thrust library (Bell and Hoberock, 2012). In general, except for some minor jittering, the performance is relatively stable.

Since the *loop* task is the most complicated one, we further investigated the experimental results, as listed in Table 6.6. Since the thread block size should not be too small, we calculate the mean and standard deviation of throughput with block size greater than or equal to 32. Heuristically, we also tested the performance for thread block sizes of 'key' configurations, which we refer to thread block sizes of 128, 256, 384, or 512. We also compare the statistics with the Thrust baseline (Bell

Table 6.6: Statistics for the *loop* tasks under various platforms and vector sizes. Best results are in boldface. Columns under 'all' cover thread block sizes no smaller than 32; 'key' columns count thread block sizes of $128, 256, 384$, or 512.

Platform	size	all		key		Thrust (Bell and Hoberock, 2012)
		mean	std	mean	std	
NX	1k	0.012	0.001	**0.013**	0.000	0.007
2080 Ti	1k	0.161	0.014	**0.177**	0.006	0.105
3090	1k	0.143	0.013	**0.152**	0.010	0.109
NX	2^{16}	0.492	0.032	**0.517**	0.035	0.129
2080 Ti	2^{16}	7.463	0.656	**8.533**	0.236	6.641
3090	2^{16}	7.610	0.566	**7.749**	0.206	6.189
NX	2^{23}	1.202	0.061	**1.285**	0.010	1.142
2080 Ti	2^{23}	18.939	1.592	21.470	0.330	**21.563**
3090	2^{23}	24.379	1.807	**27.058**	0.225	26.471

and Hoberock, 2012). The table shows that the 'key' configurations generally have higher average performance and much lower standard deviation. Except for the case with the largest vector size on the 2080 Ti platform, the average performance of key configurations also outperforms the baseline. Therefore, they serve well enough as universal configurations.

6.5.5 Summary

GPUs provide powerful parallel computing capacity for modern applications. To help programmers write GPU programs more efficiently, we propose IcyVeins, an STL-like parallel programming library. We demonstrated how interface design benefits standard programming techniques, including operator fusion and layout optimization. We also conducted comprehensive experiments to investigate how execution parameters affect program performance. Our library also outperforms the Thrust library in many test cases, showing the effectiveness of such designs. We hope to see more research works aiming to improve the accessibility of GPU programming.

Listing 6 The interface of `icyveins::views::transform`.

```
class view {};
template<typename Callable,
         typename ClosureType = view,
         typename InputIt = nullptr_t>
class transform : public view {
public:
  //re-export types
  using F = Callable;
  using C = ClosureType;
  using iterator = InputIt;
  // constructors
  constexpr transform(F &func);
  constexpr transform(F &func, C &closure);
  constexpr transform(F &func, C &closure,
            InputIt begin, InputIt end);
  // operator overload
  template <typename V, typename Closure>
  friend constexpr typename std::enable_if<
    !std::is_base_of<view, V>::value,
    transform<typename Closure::F,
              typename Closure::C,
              typename V::const_iterator>>::type
  operator|(const V &v, const Closure &closure);
  template <typename L, typename R>
  friend constexpr typename std::enable_if<
    std::is_base_of<view, L>::value,
    transform<typename R::F, L,
              typename L::iterator>>::type
  operator|(const L &lhs, const R &rhs);
private:
  F _func;
  C _closure;
  InputIt _begin;
  InputIt _end;
};
```

Listing 7 CUDA source file and the corresponding PTX assembly of operator fusion using `icyveins::views::transform`.

```
// ranges.cu
auto func = [] __device__ (int i) -> int
            { return i * i; };
auto  v = icyveins::vector<int>(100, 1);
using T = icyveins::view::transform<
            decltype(func)>;
auto view = v | T(f) | T(f) | T(f);
// ranges.ptx
...
ld.global.u32   %r5, [%rd11];
add.s32         %r6, %r5, %r5;
add.s32         %r7, %r6, %r6;
add.s32         %r8, %r7, %r7;
add.s64         %rd13, %rd9, %rd7;
st.global.u32   [%rd13], %r8;
...
```

Listing 8 A motivating example of data layout optimization.

```
//Array of Struct (AoS)
icv::vector<icv::tuple<int, double>> aos;
for(auto &t: aos) {
    icv::get<0>(t) += 1;
}

//Struct of Array (SoA)
icv::tuple<icv::vector<int>,
           icv::vector<double>> soa;
for(auto &i: icv::get<0>(soa)) {
    i += 1;
}
```

Listing 9 A simple template parameter interface for data layout switch.

```
enum class Layout { AoS, SoA };

template <Layout TLayout, typename... T>
class layout_vector {};
template <typename... T>
class layout_vector<Layout::AoS, T...>{
    vector<tuple<T...>> _v;
}
template <typename... T>
class layout_vector<Layout::SoA, T...>{
    tuple<T *...> _sv;
}
```

Listing 10 Dynamic dispatching using template parameter pack.

```
template <typename... T>
class layout_vector<Layout::SoA, T...> {
   public:
    using reference = tuple<T &...>;
    using size_type = std::size_t;
    __device__ reference
    operator[](size_type index) noexcept {
        return _get(
            index,
            std::make_integer_sequence<
                unsigned, sizeof...(T)>());
    }
   private:
    template <unsigned... Ids>
    __device__ constexpr reference
    _get(size_type index,
         std::integer_sequence<unsigned,
                               Ids...>) {
        return reference{
            icv::get<Ids>(_sv)[index]...};
    }
    tuple<T *...> _sv;
}
```

Listing 11 Data layout switch from SoA to AoS.

```
template <typename SoA, typename AoSp>
__global__ void soa2aos(const SoA soa,
                        AoSp aosp) {
    int i = blockDim.x * blockIdx.x +
            threadIdx.x;
    auto size = soa.size();
    if (i < size) {
        aosp[i] = soa[i];
    }
}
layout_vector<DataLayout::AoS, T...>
    &operator=(const layout_vector<
               DataLayout::SoA, T...> &soa) {
    soa2aos<<</*...*/>>>(soa, _v.begin());
}
```

Chapter 7

Conclusion and Future Work

In this thesis, we have proposed several algorithmic methodologies to tackle the efficiency issue in electronic design automation. Our major contributions include:

- In Chapter 3, we have proposed, for the first time, a graph learning-based scheme for arithmetic block identification. We formulate a node classification problem to identify boundaries of arithmetic components from large design netlists. To facilitate the above procedure, we developed customized GNN architecture dedicated for netlists representation learning. We further presented network-flow-based method for input-output matching. Experiments on open-source RISC-V CPU designs synthesized by industrial tools confirm the effectiveness and efficiency of our proposed framework compared with other state-of-the-art macro block matching solutions. We also discussed a fast cut enumeration algorithm.

- In Chapter 4, we have presented the first systematic attempt on leveraging the idea of reinforcement learning to floorplanning. Unlike many previous RL works that aimed at solution construction, we investigated the possibility of acquiring local search heuristics through massive search experiments. We trained an agent, which performs a walk in the search space by selecting a candidate neighbor solution at each step, using a novel deep Q-learning algorithm with action sampling. The experimental results have demonstrated the effectiveness of our proposed methods.

- In Chapter 5, we have scrutinized a wafer-scale deep learning accelerator placement problem, a case study of specific physical synthesis for advanced neural network processors. We especially argue that datapath design is an essential methodology in the above procedures due to the organized computational graph of neural networks. Experimental results show that datapath driven floorplan greatly outperforms standard methods such as simulated annealing.

- In Chapter 6, we have introduced novel parallel algorithms and implementation for design rule checking. We showed that many DRC tasks can be solved via a general prefix computation scheme, which we parallelized with both nontrivial theoretical guarantee and efficiency in practice. We implemented the algorithms on modern GPUs, resulting in two state-of-the-art design rule checkers, namely X-Check, the one integrated into KLayout, and OpenDRC, which is now open-source.

With the above exploration and discussions, we have presented novel algorithmic methodologies to improve electronic design automation efficiency. Given the importance of EDA tools and the continuous advancement of related techniques, we hope to see more research along this line. In particular, following are some directions and open problems for future research:

- Training neural models for EDA tasks is generally expensive due to the lack of data and the runtime cost of running EDA tools. There could be efforts to lessen the barrier to such training, such as constructing universal, multi-modality circuit datasets, developing model-based reinforcement learning algorithms for EDA tasks, and building explainable AI models.

- Customizing design flow for specific designs is critical. We have explored datapath driven floorplan for deep learning accelerators. It would be interesting to see how systematic investigation considering technology, design, and algorithm would benefit the whole design flow.

- Current efforts to X-Check and OpenDRC (algorithms, codebase, etc.) can be extended to other layout-centric tasks, such as Layout Versus Schematic (LVS), layout pattern generation, and design rule checking beyond digital designs (e.g., for nanophotonic or analog circuits).

References

Emile Aarts, Emile HL Aarts, and Jan Karel Lenstra. 2003. *Local Search in Combinatorial Optimization*. Princeton University Press. https://doi.org/10.1515/9780691187563

Tutu Ajayi, Vidya A. Chhabria, Mateus Fogaça, Soheil Hashemi, Abdelrahman Hosny, Andrew B. Kahng, Minsoo Kim, Jeongsup Lee, Uday Mallappa, Marina Neseem, Geraldo Pradipta, Sherief Reda, Mehdi Saligane, Sachin S. Sapatnekar, Carl Sechen, Mohamed Shalan, William Swartz, Lutong Wang, Zhehong Wang, Mingyu Woo, and Bangqi Xu. 2019. Toward an Open-Source Digital Flow: First Learnings from the Openroad Project. In *Proceedings of the 56th Annual Design Automation Conference 2019* (Las Vegas, NV, USA) *(Dac '19)*. Article 76, 4 pages. https://doi.org/10.1145/3316781.3326334

Irina Alam, Tianmu Li, Sean Brock, and Puneet Gupta. 2023. DRDebug: Automated Design Rule Debugging. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 42, 2 (2 2023), 606–615. https://doi.org/10.1109/tcad.2022.3174722

Aida Ali, Siti Mariyam Shamsuddin, and Anca Ralescu. 2015. Classification with Class Imbalance Problem: A Review. *Int. J. Advance Soft Compu. Appl* 7 (2015), 176–204.

Manoj Alwani, Han Chen, Michael Ferdman, and Peter Milder. 2016. Fused-Layer CNN Accelerators, In The 49th Annual IEEE/ACM International Symposium on Microarchitecture (Taipei, Taiwan). *Micro*, Article 22, 12 pages. https://doi.org/10.1109/micro.2016.7783725

Alon Amid, David Biancolin, Abraham Gonzalez, Daniel Grubb, Sagar Karandikar, Harrison Liew, Albert Magyar, Howard Mao, Albert Ou, Nathan Pemberton, et al. 2020. Chipyard: Integrated Design, Simulation, And Implementation Framework for Custom SOCs. *IEEE Micro* 40, 4 (7 2020), 10–21. https://doi.org/10.1109/mm.2020.2996616

Srinivasa Rao Arikati and Ravi Varadarajan. 1997. A Signature Based Approach to Regularity Extraction, In ICCAD (San Jose, CA, USA). *International Conference on Computer Aided Design* 97, 542–545. https://doi.org/10.1109/iccad.1997.643592

Krste Asanovic, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, et al. 2016. *The Rocket Chip Generator*. Technical Report. EECS Department, University of California at Berkeley.

Leonid Azriel, Ran Ginosar, and Avi Mendelson. 2019. SoK: An Overview of Algorithmic Methods in IC Reverse Engineering, In Proceedings of the 3rd ACM Workshop on Attacks and Solutions in Hardware Security Workshop (London, United Kingdom), Chip-Hong Chang, Ulrich Rührmair, Daniel E. Holcomb, and Patrick Schaumont (Eds.). *ASHES@CCS*, 65–74. https://doi.org/10.1145/3338508.3359575

Sungmin Bae, Hyung-Ock Kim, Jungyun Choi, and Jaehong Park. 2015. Coarse-Grained Structural Placement for a Synthesized Parallel Multiplier, In Proceedings of the 2015 Symposium on International Symposium on Physical Design (Monterey, California, USA), Azadeh Davoodi and Evangeline Young (Eds.). *ACM International Symposium on Physical Design*, 17–24. https://doi.org/10.1145/2717764.2717775

Nathan Bell and Jared Hoberock. 2012. Thrust: a Productivity-Oriented Library for CUDA. In *GPU Computing Gems Jade Edition*, Wen mei W. Hwu (Ed.). Morgan Kaufmann, Boston, 359–371. https://doi.org/10.1016/B978-0-12-385963-1.00026-5

Irwan Bello, Hieu Pham, Quoc V Le, Mohammad Norouzi, and Samy Bengio. 2016. Neural Combinatorial Optimization with Reinforcement Learning. *ArXiv preprint* abs/1611.09940 (11 2016). https://arxiv.org/abs/1611.09940

Dmitrii Beloborodov, AE Ulanov, Jakob N Foerster, Shimon Whiteson, and AI Lvovsky. 2020. Reinforcement Learning Enhanced Quantum-Inspired Algorithm for Combinatorial Optimization. *ArXiv preprint* abs/2002.04676 (2 2020), 25009. https://doi.org/10.1088/2632-2153/abc328

Una Benlic, Michael G. Epitropakis, and Edmund K. Burke. 2017. A Hybrid Breakout Local Search And Reinforcement Learning Approach to the Vertex Separator Problem. *European Journal of Operational Research* 261, 3 (9 2017), 803–818. https://doi.org/10.1016/j.ejor.2017.01.023

Bentley and Wood. 1980. An Optimal Worst Case Algorithm for Reporting Intersections of Rectangles. *IEEE Trans. Comput.* C-29, 7 (7 1980), 571–577. https://doi.org/10.1109/tc.1980.1675628

Jon Louis Bentley. 1975. Multidimensional Binary Search Trees Used for Associative Searching. *Commun. ACM* 18, 9 (9 1975), 509–517. https://doi.org/10.1145/361002.361007

Vaughn Betz and Jonathan Rose. 1997. VPR: a New Packing, Placement And Routing Tool for FPGA Research, In Proceedings of the 7th International Workshop on Field-Programmable Logic and Applications, Wayne Luk, Peter Y. K. Cheung, and Manfred Glesner (Eds.). *International Conference on Field-Programmable Logic and Applications* 1304, 213–222. https://doi.org/10.1007/3-540-63465-7_226

Kirti Bhanushali and W. Rhett Davis. 2015. FreePDK15: An Open-Source Predictive Process Design Kit for 15nm FinFET Technology, In Proceedings of the 2015 Symposium on International Symposium on Physical Design (Monterey, California, USA), Azadeh Davoodi and Evangeline Young (Eds.). *ACM International Symposium on Physical Design*, 165–170. https://doi.org/10.1145/2717764.2717782

G.E. Bier and A.R. Pleszkun. 1985. An Algorithm for Design Rule Checking on a Multiprocessor, In 22nd ACM/IEEE Design Automation Conference, Hillel Ofek and Lawrence A. O Neill (Eds.). *22nd ACM/IEEE Design Automation Conference*, 299–304. https://doi.org/10.1109/dac.1985.1585956

Justin A Boyan and Andrew W Moore. 1998. Learning Evaluation Functions for Global Optimization And Boolean Satisfiability, In AAAI/IAAI. *AAAI/IAAI*, 3–10.

Robert Brayton and Alan Mishchenko. 2010. Abc: An Academic Industrial-Strength Verification Tool, In Computer Aided Verification, Tayssir Touili, Byron Cook, and Paul Jackson (Eds.).

International Conference on Computer Aided Verification 6174, 24–40. https://doi.org/10.1007/978-3-642-14295-6_5

Brent and Kung. 1982. A Regular Layout for Parallel Adders. *IEEE Trans. Comput.* C-31, 3 (3 1982), 260–264. https://doi.org/10.1109/tc.1982.1675982

Joan Bruna, Wojciech Zaremba, Arthur Szlam, and Yann LeCun. 2014. Spectral Networks And Locally Connected Networks on Graphs, In 2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings, Yoshua Bengio and Yann LeCun (Eds.). *International Conference on Learning Representations* abs/1312.6203. http://arxiv.org/abs/1312.6203

Mateusz Buda, Atsuto Maki, and Maciej A. Mazurowski. 2018. A Systematic Study of the Class Imbalance Problem in Convolutional Neural Networks. *Neural Networks* 106 (10 2018), 249–259. https://doi.org/10.1016/j.neunet.2018.07.011

H. Cai, S. Note, P. Six, and H. De Man. 1991. A Data Path Layout Assembler for High Performance Dsp Circuits, In Proceedings of the 27th ACM/IEEE Design Automation Conference (Orlando, Florida, USA). *27th ACM/IEEE Design Automation Conference*, 306–311. https://doi.org/10.1145/123186.123284

A.E. Caldwell, I.L. Markov, and A.B. Kahng. 2002. Toward Cad-Ip Reuse: a Web Bookshelf of Fundamental Algorithms. *IEEE Design & Test of Computers* 19, 3 (5 2002), 70–79. https://doi.org/10.1109/mdt.2002.1003801

Calma. 1987. *GDSII Stream Format Manual*. Technical Report.

E.C. Carlson and R.A. Rutenbar. 1988. Mask Verification on the Connection Machine, In 25th ACM/IEEE, Design Automation Conference.Proceedings 1988. *25th ACM/IEEE, Design Automation Conference.Proceedings 1988.*, 134–140. https://doi.org/10.1109/dac.1988.14748

Erik C. Carlson and Rob A. Rutenbar. 1991. Design And Performance Evaluation of New Massively Parallel VLSI Mask Verification Algorithms in Jigsaw, In Proceedings of the 27th ACM/IEEE Design Automation Conference (Orlando, Florida, USA). *27th ACM/IEEE Design Automation Conference*, 253–259. https://doi.org/10.1145/123186.123268

K. M. Chandy and J. Misra. 1981. Asynchronous Distributed Simulation via a Sequence of Parallel Computations. *Commun. ACM* 24, 4 (4 1981), 198–206. https://doi.org/10.1145/358598.358613

Kyungwook Chang, Deepak Kadetotad, Yu Cao, Jae-sun Seo, and Sung Kyu Lim. 2017. Monolithic 3D Ic Designs for Low-Power Deep Neural Networks Targeting Speech Recognition, In 2017 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED). *International Symposium on Low Power Electronics and Design*, 1–6. https://doi.org/10.1109/islped.2017.8009175

Kyungwook Chang, Deepak Kadetotad, Yu Cao, Jae-Sun Seo, and Sung Kyu Lim. 2018. Power, Performance, And Area Benefit of Monolithic 3D Ics for On-Chip Deep Neural Networks Targeting Speech Recognition. *J. Emerg. Technol. Comput. Syst.* 14, 4, Article 42 (11 2018), 19 pages. https://doi.org/10.1145/3273956

Yun-Chih Chang, Yao-Wen Chang, Guang-Ming Wu, and Shu-Wei Wu. 2000. B*-Trees: A New Representation for Non-Slicing Floorplans, In Proceedings of the 37th Annual Design Automation Conference (Los Angeles, California, USA). *Proceedings - Design Automation Conference*, 458–463. https://doi.org/10.1145/337292.337541

Satrajit Chatterjee, Alan Mishchenko, and Robert Brayton. 2006. Factor Cuts, In Proceedings of the 2006 IEEE/ACM International Conference on Computer-Aided Design (San Jose, California), Soha Hassoun (Ed.). *IEEE/ACM International Conference on Computer-Aided Design*, 143–150. https://doi.org/10.1145/1233501.1233531

Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. 2002. Smote: Synthetic Minority Over-Sampling Technique. *Journal of Artificial Intelligence Research* 16 (6 2002), 321–357. https://doi.org/10.1613/jair.953

Bernard Chazelle and Herbert Edelsbrunner. 1992. An Optimal Algorithm for Intersecting Line Segments in the Plane. *J. Acm* 39, 1 (10 1992), 1–54. https://doi.org/10.1145/147508.147511

Deming Chen and Jason Cong. 2004. DAOmap: a Depth-Optimal Area Optimization Mapping Algorithm for FPGA Designs, In IEEE/ACM International Conference on Computer Aided Design, 2004. ICCAD-2004. *IEEE/ACM International Conference on Computer Aided Design, 2004. ICCAD-2004.*, 752–759. https://doi.org/10.1109/iccad.2004.1382677

Gengjie Chen, Chak-Wa Pui, Haocheng Li, and Evangeline F. Y. Young. 2020. Dr. Cu: Detailed Routing by Sparse Grid Graph And Minimum-Area-Captured Path Search. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 9 (1 2020), 1902–1915. https://doi.org/10.1109/tcad.2019.2927542

Tianqi Chen, Lianmin Zheng, Eddie Q. Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. Learning to Optimize Tensor Programs, In Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada, Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett (Eds.). *Neural Information Processing Systems*, 3393–3404. https://proceedings.neurips.cc/paper/2018/hash/8b5700012be65c9da25f49408d959ca0-Abstract.html

Tung-Chieh Chen and Yao-Wen Chang. 2005. Modern Floorplanning Based on Fast Simulated Annealing, In Proceedings of the 2005 International Symposium on Physical Design (San Francisco, California, USA), Patrick Groeneveld and Louis Scheffer (Eds.). *ACM International Symposium on Physical Design*, 104–112. https://doi.org/10.1145/1055137.1055161

Xinyun Chen and Yuandong Tian. 2019. Learning to Perform Local Rewriting for Combinatorial Optimization, In Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada, Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché Buc, Emily B. Fox, and Roman Garnett (Eds.). *Neural Information Processing Systems*, 6278–6289. https://proceedings.neurips.cc/paper/2019/hash/131f383b434fdf48079bff1e44e2d9a5-Abstract.html

Chung-Kuan Cheng, Andrew B. Kahng, Ilgweon Kang, and Lutong Wang. 2019. Replace: Advancing Solution Quality And Routability Validation in Global Placement. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38, 9 (9 2019), 1717–1730. https://doi.org/10.1109/tcad.2018.2859220

Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. Cudnn: Efficient Primitives for Deep Learning. *arXiv preprint arXiv:1410.0759* abs/1410.0759 (10 2014). http://arxiv.org/abs/1410.0759

Ping Chi, Shuangchen Li, Cong Xu, Tao Zhang, Jishen Zhao, Yongpan Liu, Yu Wang, and Yuan Xie. 2016. Prime: a Novel Processing-In-Memory Architecture for Neural Network Computation in Reram-Based Main Memory. *International Symposium on Computer Architecture* (6 2016), 27–39. https://doi.org/10.1109/isca.2016.13

Sheng Chou, Meng-Kai Hsu, and Yao-Wen Chang. 2012. Structure-Aware Placement for Datapath-Intensive Circuit Designs, In Proceedings of the 49th Annual Design Automation Conference (San Francisco, California), Patrick Groeneveld, Donatella Sciuto, and Soha Hassoun (Eds.). *DAC Design Automation Conference 2012*, 762–767. https://doi.org/10.1145/2228360.2228498

A. Chowdhary, S. Kale, P.K. Saripella, N.K. Sehgal, and R.K. Gupta. 1999. Extraction of Functional Regularity in Datapath Circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 18, 9 (9 1999), 1279–1296. https://doi.org/10.1109/43.784120

Chris Chu and Yiu-Chung Wong. 2008. Flute: Fast Lookup Table Based Rectilinear Steiner Minimal Tree Algorithm for VLSI Design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 27, 1 (2008), 70–83. https://doi.org/10.1109/tcad.2007.907068

Lawrence T. Clark, Vinay Vashishtha, Lucian Shifren, Aditya Gujja, Saurabh Sinha, Brian Cline, Chandarasekaran Ramamurthy, and Greg Yeric. 2016. Asap7: a 7-Nm Finfet Predictive Process Design Kit. *Microelectronics Journal* 53 (7 2016), 105–115. https://doi.org/10.1016/j.mejo.2016.04.006

J. Cong and Yuzheng Ding. 1994. Flowmap: An Optimal Technology Mapping Algorithm for Delay Optimization in Lookup-Table Based FPGA Designs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 13, 1 (1994), 1–12. https://doi.org/10.1109/43.273754

Jason Cong, Michail Romesis, and Joseph R. Shinnerl. 2005. Fast Floorplanning by Look-Ahead Enabled Recursive Bipartitioning. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 25 (1 2005), 1119–1122. https://doi.org/10.1145/1120725.1120838

Jason Cong, Chang Wu, and Yuzheng Ding. 1999. Cut Ranking And Pruning: Enabling a General And Efficient FPGA Mapping Solution, In Proceedings of the 1999 ACM/SIGDA Seventh International Symposium on Field Programmable Gate Arrays (Monterey, California, USA). *Symposium on Field Programmable Gate Arrays*, 29–35. https://doi.org/10.1145/296399.296425

Vito Dai, Luigi Capodieci, Jie Yang, and Norma Rodriguez. 2009. Developing DRC Plus Rules through 2D Pattern Extraction And Clustering Techniques. In *Design for Manufacturability through Design-Process Integration III*, Vivek K. Singh and Michael L. Rieger (Eds.), Vol. 7275. International Society for Optics and Photonics, 727517. https://doi.org/10.1117/12.814347

Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. 2016. Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering, In Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain, Daniel D. Lee, Masashi Sugiyama, Ulrike von Luxburg, Isabelle Guyon, and Roman Garnett (Eds.). *NIPS*, 3837–3845. https://proceedings.neurips.cc/paper/2016/hash/04df4d434d481c5bb723be1b6df1ee65-Abstract.html

T. Doom, J. White, A. Wojcik, and G. Chisholm. 1998. Identifying High-Level Components in Combinational Circuits, In Proceedings of the 8th Great Lakes Symposium on VLSI (Cat. No.98TB100222). *Proceedings of the 8th Great Lakes Symposium on VLSI (Cat. No.98TB100222)*, 313–318. https://doi.org/10.1109/glsv.1998.665284

Gabriel Dulac-Arnold, Richard Evans, Hado van Hasselt, Peter Sunehag, Timothy Lillicrap, Jonathan Hunt, Timothy Mann, Theophane Weber, Thomas Degris, and Ben Coppin. 2015. Deep Reinforcement Learning in Large Discrete Action Spaces. *ArXiv preprint* abs/1512.07679 (12 2015). https://arxiv.org/abs/1512.07679

Charles Elkan. 2001. The Foundations of Cost-Sensitive Learning, In Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence, IJCAI 2001, Seattle, Washington, USA, August 4-10, 2001, Bernhard Nebel (Ed.). *International Joint Conference on Artificial Intelligence* 17, 1, 973–978.

Arash Fayyazi, Soheil Shababi, Pierluigi Nuzzo, Shahin Nazarian, and Massoud Pedram. 2019. Deep Learning-Based Circuit Recognition Using Sparse Mapping And Level-Dependent Decaying Sum Circuit Representations, In 2019 Design, Automation & Test in Europe Conference & Exhibition (DATE). *Design, Automation and Test in Europe*, 638–641. https://doi.org/10.23919/date.2019.8715251

Raphael A Finkel and Jon Louis Bentley. 1974. Quad Trees a Data Structure for Retrieval on Composite Keys. *Acta informatica* 4, 1 (3 1974), 1–9. https://doi.org/10.1007/bf00288933

Luis Francisco, Tanmay Lagare, Arpit Jain, Somal Chaudhary, Madhura Kulkarni, Divya Sardana, W Rhett Davis, and Paul Franzon. 2020. Design Rule Checking with a CNN Based Feature Extractor, In MLCAD '20: 2020 ACM/IEEE Workshop on Machine Learning for CAD, Virtual Event, Iceland, November 16-20, 2020, Ulf Schlichtmann, Raviv Gal, Hussam Amrouch, and Hai (Helen) Li (Eds.). *Workshop on Machine Learning for CAD*, 9–14. https://doi.org/10.1145/3380446.3430625

Scott Fujimoto, Herke van Hoof, and David Meger. 2018. Addressing Function Approximation Error in Actor-Critic Methods, In Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018, Jennifer G. Dy and Andreas Krause (Eds.). *International Conference on Machine Learning* 80, 1582–1591. http://proceedings.mlr.press/v80/fujimoto18a.html

Mingyu Gao, Jing Pu, Xuan Yang, Mark Horowitz, and Christos Kozyrakis. 2017. Tetris: Scalable And Efficient Neural Network Acceleration with 3D Memory, In Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (Xi'an, China), Yunji Chen, Olivier Temam, and John Carter (Eds.). *International Conference on Architectural Support for Programming Languages and Operating Systems*, 751–764. https://doi.org/10.1145/3037697.3037702

Adria Gascón, Pramod Subramanyan, Bruno Dutertre, Ashish Tiwari, Dejan Jovanović, and Sharad Malik. 2014. Template-Based Circuit Understanding, In 2014 Formal Methods in Computer-Aided Design (FMCAD). *Formal Methods in Computer-Aided Design*, 83–90. https://doi.org/10.1109/fmcad.2014.6987599

F Gregoretti and Z Segall. 1984. Analysis And Evaluation of VLSI Design Rule Checking Implementation in a Multiprocessor. In *Proc. Int. Conf. Parallel Processing*. 7–14.

Jiaqi Gu, Zheng Zhao, Chenghao Feng, Mingjie Liu, Ray T Chen, and David Z Pan. 2020. Towards Area-Efficient Optical Neural Networks: An Fft-Based Architecture, In 25th Asia and South Pacific Design Automation Conference, ASP-DAC 2020, Beijing, China, January 13-16, 2020. *Asia and South Pacific Design Automation Conference*, 476–481. https://doi.org/10.1109/asp-dac47756.2020.9045156

Shixiang Gu, Ethan Holly, Timothy Lillicrap, and Sergey Levine. 2017. Deep Reinforcement Learning for Robotic Manipulation with Asynchronous Off-Policy Updates, In 2017 IEEE international conference on robotics and automation (ICRA). *IEEE International Conference on Robotics and Automation*, 3389–3396. https://doi.org/10.1109/icra.2017.7989385

Guannan Guo, Tsung-Wei Huang, Yibo Lin, and Martin Wong. 2021a. GPU-Accelerated Path-Based Timing Analysis, In 2021 58th ACM/IEEE Design Automation Conference (DAC). *Design Automation Conference*, 721–726. https://doi.org/10.1109/dac18074.2021.9586316

Pei-Ning Guo, Chung-Kuan Cheng, and Takeshi Yoshimura. 1999. An O-Tree Representation of Non-Slicing Floorplan And Its Applications, In Proceedings 1999 Design Automation Conference (Cat. No. 99CH36361). *Proceedings - Design Automation Conference*, 268–273. https://doi.org/10.1145/309847.309928

Zizheng Guo, Tsung-Wei Huang, and Yibo Lin. 2020. GPU-Accelerated Static Timing Analysis, In Proceedings of the 39th International Conference on Computer-Aided Design. *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 1–9. https://doi.org/10.1145/3400302.3415631

Zizheng Guo, Jing Mai, and Yibo Lin. 2021b. Ultrafast Cpu/GPU Kernels for Density Accumulation in Placement, In 2021 58th ACM/IEEE Design Automation Conference (DAC). *Design Automation Conference*, 1123–1128. https://doi.org/10.1109/dac18074.2021.9586149

Antonin Guttman. 1984. R-Trees: a Dynamic Index Structure for Spatial Searching, In SIGMOD 84, Proceedings of Annual Meeting, Boston, Massachusetts, June 18-21, 1984, Beatrice Yormark (Ed.). *ACM SIGMOD Conference*, 47–57. https://doi.org/10.1145/971697.602266

Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. 2018. Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor, In Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018, Jennifer G. Dy and Andreas Krause (Eds.). *International Conference on Machine Learning* 80, 1856–1865. http://proceedings.mlr.press/v80/haarnoja18b.html

Aric Hagberg, Pieter Swart, and Daniel S Chult. 2008. *Exploring Network Structure, Dynamics, And Function Using Networkx*. Technical Report. Los Alamos National Lab.(LANL), Los Alamos, NM (United States).

William L. Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive Representation Learning on Large Graphs, In Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA, Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett (Eds.). *NIPS*, 1024–1034. https://proceedings.neurips.cc/paper/2017/hash/5dd9db5e033da9c6fb5ba83c7a7ebea9-Abstract.html

Youbiao He and Forrest Sheng Bao. 2020. Circuit Routing Using Monte Carlo Tree Search And Deep Neural Networks. *ArXiv preprint* abs/2006.13607 (6 2020). https://arxiv.org/abs/2006.13607

Zhuolun He, Peiyu Liao, Siting Liu, Yuzhe Ma, Yibo Lin, and Bei Yu. 2021. Physical Synthesis for Advanced Neural Network Processors, In ASPDAC '21: 26th Asia and South Pacific Design Automation Conference, Tokyo, Japan, January 18-21, 2021. *Asia and South Pacific Design Automation Conference*, 833–840. https://doi.org/10.1145/3394885.3431625

Zhuolun He, Yuzhe Ma, and Bei Yu. 2022. X-Check: GPU-Accelerated Design Rule Checking via Parallel Sweepline Algorithms, In Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design, ICCAD 2022, San Diego, California, USA, 30 October 2022 - 3 November 2022, Tulika Mitra, Evangeline Young, and Jinjun Xiong (Eds.). *2022 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 1–9. https://doi.org/10.1145/3508352.3549383

Troels Henriksen, Niels GW Serup, Martin Elsman, Fritz Henglein, and Cosmin E Oancea. 2017. Futhark: Purely Functional GPU-Programming with Nested Parallelism And In-Place Array Updates, In Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017, Albert Cohen 0001 and Martin T. Vechev (Eds.). *ACM-SIGPLAN Symposium on Programming Language Design and Implementation*, 556–571. https://doi.org/10.1145/3062341.3062354

Matteo Hessel, Joseph Modayil, Hado van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Gheshlaghi Azar, and David Silver. 2018. Rainbow: Combining Improvements in Deep Reinforcement Learning, In Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018, Sheila A. McIlraith and Kilian Q. Weinberger (Eds.). *AAAI Conference on Artificial Intelligence*, 3215–3222. https://doi.org/10.1609/aaai.v32i1.11796

Kai-Ti Hsu, Subarna Sinha, Yu-Chuan Pi, Charles Chiang, and Tsung-Yi Ho. 2011. A Distributed Algorithm for Layout Geometry Operations, In Proceedings of the 48th Design Automation Conference, DAC 2011, San Diego, California, USA, June 5-10, 2011, Leon Stok, Nikil D. Dutt, and Soha Hassoun (Eds.). *Design Automation Conference*, 182–187. https://doi.org/10.1145/2024724.2024765

Chau-Chin Huang, Bo-Qiao Lin, Hsin-Ying Lee, Yao-Wen Chang, Kuo-Sheng Wu, and Jun-Zhi Yang. 2017. Graph-Based Logic Bit Slicing for Datapath-Aware Placement, In Proceedings of the 54th Annual Design Automation Conference, DAC 2017, Austin, TX, USA, June 18-22, 2017. *Design Automation Conference*, 1–6. https://doi.org/10.1145/3061639.3062254

Tsung-Wei Huang, Dian-Lun Lin, Chun-Xun Lin, and Yibo Lin. 2021. Taskflow: a Lightweight Parallel And Heterogeneous Task Graph Computing System. *IEEE Transactions on Parallel and Distributed Systems* 33, 6 (4 2021), 1303–1320. https://doi.org/10.1109/tpds.2021.3104255

Tsung-Wei Huang and Martin DF Wong. 2015. Opentimer: a High-Performance Timing Analysis Tool, In Proceedings of the IEEE/ACM International Conference on Computer-Aided Design, ICCAD 2015, Austin, TX, USA, November 2-6, 2015, Diana Marculescu and Frank Liu (Eds.). *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 895–902. https://doi.org/10.1109/iccad.2015.7372666

Daniele Ielmini and Stefano Ambrogio. 2019. Emerging Neuromorphic Devices. *Nanotechnology* 31, 9 (11 2019), 092001. https://doi.org/10.1088/1361-6528/ab554b

Paolo Ienne and Alexander Grießing. 1998. Practical Experiences with Standard-Cell Based Datapath Design Tools. Do We Really Need Regular Layouts?. In *Proceedings 1998 Design and Automation Conference. 35th DAC. (Cat. No.98CH36175)*. IEEE, 396–401. https://doi.org/10.1109/dac.1998.724505

ISO/IEC TS 23619:2021 2021. *Information Technology — C++ Extensions for Reflection*. Standard. International Organization for Standardization. https://doi.org/10.3403/30384531u

T. Jaakkola, M. I. Jordan, and S. P. Singh. 1994. On the Convergence of Stochastic Iterative Dynamic Programming Algorithms. *Neural Computation* 6, 6 (8 1994), 1185–1201. https://doi.org/10.1162/neco.1994.6.6.1185

Michael James, Marvin Tom, Patrick Groeneveld, and Vladimir Kibardin. 2020. Ispd 2020 Physical Mapping of Neural Networks on a Wafer-Scale Deep Learning Accelerator, In Proceedings of the 2020 International Symposium on Physical Design, William Swartz and Jens Lienig (Eds.). *ACM International Symposium on Physical Design*, 145–149. https://doi.org/10.1145/3372780.3380846

Joe Jeddeloh and Brent Keeth. 2012. Hybrid Memory Cube New Dram Architecture Increases Density And Performance. In *Symposium on VLSI Technology*. IEEE, 87–88. https://doi.org/10.1109/vlsit.2012.6242474

Bentian Jiang, Jingsong Chen, Jinwei Liu, Lixin Liu, Fangzhou Wang, Xiaopeng Zhang, and Evangeline FY Young. 2020. Cu. Poker: Placing DNNs on Wafer-Scale Al Accelerator with Optimal Kernel Sizing, In IEEE/ACM International Conference on Computer Aided Design, ICCAD 2020, San Diego, CA, USA, November 2-5, 2020. *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 1–9. https://doi.org/10.1145/3400302.3415688

Tong Jing, Xian-Long Hong, Yi-Ci Cai, Jing-Yu Xu, Chang-Qi Yang, Yi-Qian Zhang, Qiang Zhou, and Weimin Wu. 2002. Data-Path Layout Design Inside Soc. In *IEEE 2002 International Conference on Communications, Circuits and Systems and West Sino Expositions*, Vol. 2. 1406–1410. https://doi.org/10.1109/icccas.2002.1179043

Joseph JéJé. 1992. An Introduction to Parallel Algorithms. *Reading, MA: Addison-Wesley* 10 (1992), 133889.

Andrew B Kahng, Lutong Wang, and Bangqi Xu. 2018. Tritonroute: An Initial Detailed Router for Advanced VLSI Technologies. In *Proceedings of the International Conference on Computer-Aided Design*, Iris Bahar (Ed.). 1–8. https://doi.org/10.1145/3240765.3240766

Rajiv Kane and Sartaj Sahni. 1984. A Systolic Design Rule Checker, In Proceedings of the 21st Design Automation Conference, DAC '84, Albuquerque, New Mexico, June 25-27, 1984, Patricia H. Lambert, Hillel Ofek, Lawrence A. O'Neill, Pat O. Pistilli, Paul Losleben, J. D. Nash, Dennis W. Shaklee, Bryan T. Preas, and Harvey N. Lerman (Eds.). *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 6, 243–250. https://doi.org/10.1109/tcad.1987.1270242

Bingyi Kang, Saining Xie, Marcus Rohrbach, Zhicheng Yan, Albert Gordo, Jiashi Feng, and Yannis Kalantidis. 2020. Decoupling Representation And Classifier for Long-Tailed Recognition, In 8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020. *International Conference on Learning Representations* abs/1910.09217. https://openreview.net/forum?id=r1gRTCVFvB

Elias B. Khalil, Hanjun Dai, Yuyu Zhang, Bistra Dilkina, and Le Song. 2017. Learning Combinatorial Optimization Algorithms Over Graphs, In Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA, Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett (Eds.). *NIPS* abs/1704.01665, 6348–6358. https://proceedings.neurips.cc/paper/2017/hash/d9896106ca98d3d05b8cbdf4fd8b13a1-Abstract.html

Boyeal Kim, Sang Hyun Lee, Hyun Kim, Duy-Thanh Nguyen, Minh-Son Le, Ik Joon Chang, Dohun Kwon, Jin Hyeok Yoo, Jun Won Choi, and Hyuk-Jae Lee. 2020. Pcm: Precision-Controlled Memory System for Energy Efficient Deep Neural Network Training, In 2020 Design, Automation & Test in Europe Conference & Exhibition, DATE 2020, Grenoble, France, March 9-13, 2020. *Design, Automation and Test in Europe*, 1199–1204. https://doi.org/10.23919/date48585.2020.9116530

Diederik P. Kingma and Jimmy Ba. 2015. Adam: a Method for Stochastic Optimization, In 3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings, Yoshua Bengio and Yann LeCun (Eds.). *International Conference on Learning Representations* abs/1412.6980. http://arxiv.org/abs/1412.6980

Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks, In 5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings. *International Conference on Learning Representations* abs/1609.02907. https://openreview.net/forum?id=SJU4ayYgl

Scott Kirkpatrick, C Daniel Gelatt, and Mario P Vecchi. 1983. Optimization by Simulated Annealing. *science* 220, 4598 (5 1983), 671–680. https://doi.org/10.1126/science.220.4598.671

Matthias Köfferlein. 2018. *KLayout*. Technical Report.

Vijay R Konda and John N Tsitsiklis. 2000. Actor-Critic Algorithms, In Advances in Neural Information Processing Systems 12, [NIPS Conference, Denver, Colorado, USA, November 29 - December 4, 1999], Sara A. Solla, Todd K. Leen, and Klaus-Robert Müller (Eds.). *NIPS*, 1008–1014. https://www.semanticscholar.org/paper/ac4af1df88e178386d782705acc159eaa0c3904a

Wouter Kool, Herke van Hoof, and Max Welling. 2019. Attention, Learn to Solve Routing Problems!, In 7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019. *International Conference on Learning Representations*. https://openreview.net/forum?id=ByxBFsRqYm

Sandeep Koranne. 2004. A High Performance Simd Framework for Design Rule Checking on Sony'S Playstation 2 Emotion Engine Platform [Ic Layout], In International Symposium on Signals, Circuits and Systems. Proceedings, SCS 2003.(Cat. No. 03EX720). *International Symposium on Signals, Circuits and Systems. Proceedings, SCS 2003. (Cat. No.03EX720)*, 371–376. https://doi.org/10.1109/isqed.2004.1283702

Thomas Kutzschebauch. 2000. Efficient Logic Optimization Using Regularity Extraction, In ICCD. *Proceedings 2000 International Conference on Computer Design*, 487–493. https://doi.org/10.1109/iccd.2000.878327

Thomas Kutzschebauch and Leon Stok. 2000. Regularity Driven Logic Synthesis, In Proceedings of the 2000 IEEE/ACM International Conference on Computer-Aided Design, 2000, San Jose, California, USA, November 5-9, 2000, Ellen Sentovich (Ed.). *IEEE/ACM International Conference on Computer Aided Design. ICCAD - 2000. IEEE/ACM Digest of Technical Papers (Cat. No.00CH37140)*, 439–446. https://doi.org/10.1109/iccad.2000.896511

Ulrich Lauther. 1981. An O (N Log N) Algorithm for Boolean Mask Operations, In Proceedings of the 18th Design Automation Conference, DAC '81, Nashville, Tennessee, USA, June 29 - July 1, 1981, Robert J. Smith II (Ed.). *Design Automation Conference*, 555–562. https://doi.org/10.1109/dac.1981.1585410

Dong Uk Lee, Kyung Whan Kim, Kwan Weon Kim, Hongjung Kim, Ju Young Kim, Young Jun Park, Jae Hwan Kim, Dae Suk Kim, Heat Bit Park, Jin Wook Shin, Jang Hwan Cho, Ki Hun Kwon, Min Jeong Kim, Jaejin Lee, Kun Woo Park, Byongtae Chung, and Sungjoo Hong. 2014. 25.2 A 1.2V 8Gb 8-channel 128GB/s high-bandwidth memory (HBM) stacked DRAM with effective microbump I/O test methods using 29nm process and TSV. In *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*. 432–433. https://doi.org/10.1109/ISSCC.2014.6757501

Can Li, Daniel Belkin, Yunning Li, Peng Yan, Miao Hu, Ning Ge, Hao Jiang, Eric Montgomery, Peng Lin, Zhongrui Wang, et al. 2018. Efficient And Self-Adaptive In-Situ Learning in Multilayer Memristor Neural Networks. *Nature communications* 9, 1 (6 2018), 1–8. https://doi.org/10.1038/s41467-018-04484-2

Haocheng Li, Satwik Patnaik, Abhrajit Sengupta, Haoyu Yang, Johann Knechtel, Bei Yu, Evangeline F.Y. Young, and Ozgur Sinanoglu. 2019. Attacking Split Manufacturing from a Deep Learning Perspective, In Proceedings of the 56th Annual Design Automation Conference 2019 (Las Vegas, NV, USA). *Design Automation Conference*, Article 135, 6 pages. https://doi.org/10.1145/3316781.3317780

Jiankun Li, Chen Ge, Jianyu Du, Can Wang, Guozhen Yang, and Kuijuan Jin. 2020. Reproducible Ultrathin Ferroelectric Domain Switching for High-Performance Neuromorphic Computing. *Advances in Materials* 32, 7 (12 2020), 1905764. https://doi.org/10.1002/adma.201905764

Nan Li and Elena Dubrova. 2011. Aig Rewriting Using 5-Input Cuts, In 2011 IEEE 29th International Conference on Computer Design (ICCD). *ICCD*, 429–430. https://doi.org/10.1109/iccd.2011.6081434

Wenchao Li, Adria Gascon, Pramod Subramanyan, Wei Yang Tan, Ashish Tiwari, Sharad Malik, Natarajan Shankar, and Sanjit A Seshia. 2013. Wordrev: Finding Word-Level Structures in a Sea of Bit-Level Gates, In 2013 IEEE International Symposium on Hardware-Oriented Security

and Trust, HOST 2013, Austin, TX, USA, June 2-3, 2013. *IEEE International Symposium on Hardware Oriented Security and Trust*, 67–74. https://doi.org/10.1109/hst.2013.6581568

Haiguang Liao, Qingyi Dong, Xuliang Dong, Wentai Zhang, Wangyang Zhang, Weiyi Qi, Elias Fallon, and Levent Burak Kara. 2020a. Attention Routing: Track-Assignment Detailed Routing Using Attention-Based Reinforcement Learning. *ArXiv preprint* abs/2004.09473 (4 2020). https://doi.org/10.1115/detc2020-22219

Haiguang Liao, Wentai Zhang, Xuliang Dong, Barnabas Poczos, Kenji Shimada, and Levent Burak Kara. 2020b. A Deep Reinforcement Learning Approach for Global Routing. *Journal of Mechanical Design* 142, 6 (6 2020). https://doi.org/10.1115/1.4045044

Chang-Tzu Lin, De-Sheng Chen, and Yi-Wen Wang. 2002. An Efficient Genetic Algorithm for Slicing Floorplan Area Optimization. In *2002 IEEE International Symposium on Circuits and Systems. Proceedings (Cat. No. 02CH37353)*, Vol. 2. IEEE, Ii–ii. https://www.semanticscholar.org/paper/7187953fc5990365e0e778f9b4a2f11acc71c1ac

Long-Ji Lin. 1992. Self-Improving Reactive Agents Based on Reinforcement Learning, Planning And Teaching. *Machine learning* 8, 3-4 (5 1992), 293–321. https://doi.org/10.1007/978-1-4615-3618-5_5

Shiju Lin, Jinwei Liu, and Martin DF Wong. 2021. Gamer: GPU Accelerated Maze Routing, In 2021 IEEE/ACM International Conference on Computer Aided Design (ICCAD). *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 42, 1–8. https://doi.org/10.1109/tcad.2022.3184281

Yibo Lin. 2020. GPU Acceleration in VLSI Back-End Design: Overview And Case Studies, In 2020 IEEE/ACM International Conference on Computer Aided Design (ICCAD). *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 1–4. https://doi.org/10.1145/3400302.3415765

Yibo Lin, Shounak Dhar, Wuxi Li, Haoxing Ren, Brucek Khailany, and David Z Pan. 2019. Dreamplace: Deep Learning Toolkit-Enabled GPU Acceleration for Modern VLSI Placement, In Proceedings of the 56th Annual Design Automation Conference 2019. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 1–6. https://doi.org/10.1145/3316781.3317803

Yibo Lin, Zixuan Jiang, Jiaqi Gu, Wuxi Li, Shounak Dhar, Haoxing Ren, Brucek Khailany, and David Z Pan. 2020. Dreamplace: Deep Learning Toolkit-Enabled GPU Acceleration for Modern VLSI Placement. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 40, 4 (6 2020), 748–761. https://doi.org/10.1145/3316781.3317803

Andrew C Ling, Jianwen Zhu, and Stephen D Brown. 2007. Bddcut: Towards Scalable Symbolic Cut Enumeration, In 2007 Asia and South Pacific Design Automation Conference. *Asia and South Pacific Design Automation Conference*, 408–413. https://doi.org/10.1109/aspdac.2007.358020

Chunsen Liu, Huawei Chen, Shuiyuan Wang, Qi Liu, Yu-Gang Jiang, David Wei Zhang, Ming Liu, and Peng Zhou. 2020. Two-Dimensional Materials for Next-Generation Computing Technologies. *Nature Nanotechnology* 15, 7 (7 2020), 545–557. https://doi.org/10.1038/s41565-020-0724-3

Siting Liu, Peiyu Liao, Rui Zhang, Zhitang Chen, Wenlong Lv, Yibo Lin, and Bei Yu. 2022. Fastgr: Global Routing on Cpu-GPU with Heterogeneous Task Graph Scheduler. *IEEE/ACM Proceedings Design, Automation and Test in Europe (DATE)* (3 2022), 760–765. https://doi.org/10.1109/tcad.2022.3217668

Jingwei Lu, Pengwen Chen, Chin-Chih Chang, Lu Sha, Dennis J-H Huang, Chin-Chi Teng, and Chung-Kuan Cheng. 2014. Eplace: Electrostatics Based Placement Using Nesterov'S Method, In The 51st Annual Design Automation Conference 2014, DAC '14, San Francisco, CA, USA, June 1-5, 2014. *Design Automation Conference*, 1–6. https://doi.org/10.1145/2593069.2593133

Zhen Luo, Margaret Martonosi, and Pranav Ashar. 2000. An Edge-Endpoint-Based Configurable Hardware Architecture for VLSI Layout Design Rule Checking. *VLSI Design* 10, 3 (2000), 249–263. https://doi.org/10.1155/2000/71046

Yuzhe Ma. 2020. *Methodologies for efficient hardware design automation and hardware-friendly learning*. Ph. D. Dissertation. The Chinese University of Hong Kong (Hong Kong).

Yufei Ma, Yu Cao, Sarma Vrudhula, and Jae-sun Seo. 2017. Optimizing Loop Operation And Dataflow in FPGA Acceleration of Deep Convolutional Neural Networks, In Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA 2017, Monterey, CA, USA, February 22-24, 2017, Jonathan W. Greene and Jason Helge Anderson (Eds.). *Symposium on Field Programmable Gate Arrays*, 45–54. https://doi.org/10.1145/3020078.3021736

Ky MacPherson. 1995. *Parallel Algorithms for Layout Verification*. Master's thesis. Citeseer. https://doi.org/10.1006/jpdc.1996.0096

Alireza Mahzoon, Daniel Große, and Rolf Drechsler. 2018. Polycleaner: Clean Your Polynomials Before Backward Rewriting to Verify Million-Gate Multipliers. In *Proceedings of the International Conference on Computer-Aided Design*, Iris Bahar (Ed.). 1–8. https://doi.org/10.1145/3240765.3240837

Alireza Mahzoon, Daniel Große, and Rolf Drechsler. 2019. Revsca: Using Reverse Engineering to Bring Light into Backward Rewriting for Big And Dirty Multipliers. In *Proceedings of the 56th Annual Design Automation Conference 2019*. 1–6. https://doi.org/10.1145/3316781.3317898

James Malcolm, Pavan Yalamanchili, Chris McClanahan, Vishwanath Venugopalakrishnan, Krunal Patel, and John Melonakos. 2012. Arrayfire: a GPU Acceleration Platform. In *Modeling and simulation for defense systems and applications VII*, Vol. 8403. 84030a. https://doi.org/10.1117/12.921122

Joshua David Marantz. 1986. Exploiting Parallelism in VLSI Cad. *Proc. IEEE ICCD'86* (1986).

Edward M McCreight. 1980. *Efficient Algorithms for Enumerating Intersecting Intervals And Rectangles*. Technical Report.

Trevor L McDonell, Manuel MT Chakravarty, Gabriele Keller, and Ben Lippmeier. 2013. Optimising Purely Functional GPU Programs. *ACM SIGPLAN Notices* 48, 9 (9 2013), 49–60. https://doi.org/10.1145/2500365.2500595

Travis Meade, Shaojie Zhang, Zheng Zhao, David Pan, and Yier Jin. 2016. Gate-Level Netlist Reverse Engineering Tool Set for Functionality Recovery And Malicious Logic Detection. In *Proc. ISTFA*. https://doi.org/10.31399/asm.cp.istfa2016p0342

Azalia Mirhoseini, Anna Goldie, Mustafa Yazgan, Joe Jiang, Ebrahim Songhori, Shen Wang, Young-Joon Lee, Eric Johnson, Omkar Pathak, Sungmin Bae, et al. 2020. Chip Placement with Deep Reinforcement Learning. *ArXiv preprint* abs/2004.10746 (4 2020). https://arxiv.org/abs/2004.10746

Alan Mishchenko, Satrajit Chatterjee, and Robert Brayton. 2006. Dag-Aware Aig Rewriting: a Fresh Look at Combinational Logic Synthesis, In 2006 43rd ACM/IEEE Design Automation Conference, Ellen Sentovich (Ed.). *2006 43rd ACM/IEEE Design Automation Conference*, 532–535. https://doi.org/10.1145/1146909.1147048

Alan Mishchenko, Satrajit Chatterjee, and Robert K Brayton. 2007a. Improvements to Technology Mapping for Lut-Based FPGAs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 26, 2 (2 2007), 240–253. https://doi.org/10.1145/1117201.1117208

Alan Mishchenko, Sungmin Cho, Satrajit Chatterjee, and Robert Brayton. 2007b. Combinational And Sequential Mapping with Priority Cuts, In 2007 IEEE/ACM International Conference on Computer-Aided Design, Georges G. E. Gielen (Ed.). *IEEE/ACM International Conference on Computer-Aided Design*, 354–361. https://doi.org/10.1109/iccad.2007.4397290

Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. 2016. Asynchronous Methods for Deep Reinforcement Learning, In Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016, Maria-Florina Balcan and Kilian Q. Weinberger (Eds.). *International Conference on Machine Learning* 48, 1928–1937. http://proceedings.mlr.press/v48/mniha16.html

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. 2013. Playing Atari with Deep Reinforcement Learning. *arXiv preprint arXiv:1312.5602* abs/1312.5602 (12 2013). http://arxiv.org/abs/1312.5602

Aaftab Munshi. 2009. The OpenCL Specification, In 2009 IEEE Hot Chips 21 Symposium (HCS). *IEEE Hot Chips Symposium*, 1–314. https://doi.org/10.1109/hotchips.2009.7478342

Hiroshi Murata, Kunihiro Fujiyoshi, Shigetoshi Nakatake, and Yoji Kajitani. 2003. Rectangle-Packing-Based Module Placement. In *The Best of ICCAD*, Richard L. Rudell (Ed.). Springer, 535–548. https://doi.org/10.1007/978-1-4615-0292-0_42

Kevin E Murray and Vaughn Betz. 2019. Adaptive FPGA Placement Optimization via Reinforcement Learning. In *Workshop on Machine Learning for CAD*. 1–6. https://doi.org/10.1109/mlcad48534.2019.9142079

SK Nandy. 1994. Geometric Design Rule Check of VLSI Layouts in Distributed Computing Environment. *VLSI design (Print)* 1, 2 (1994), 155–167. https://doi.org/10.1155/1994/54126

Alexander Nareyek. 2003. Choosing Search Heuristics by Non-Stationary Reinforcement Learning. In *Metaheuristics: Computer decision-making*. Springer, 523–544. https://doi.org/10.1007/978-1-4757-4137-7_25

Maxim Naumov, L Chien, Philippe Vandermersch, and Ujval Kapasi. 2010. Cusparse Library. In *GPU Technology Conference*.

John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. 2008. Scalable Parallel Programming with CUDA: Is CUDA the Parallel Programming Model That Application Developers Have Been Waiting For? *Queue* 6, 2 (3 2008), 40–53. https://doi.org/10.1145/1365490.1365500

Raymond XT Nijssen and Jochen AG Jess. 1996. Two-Dimensional Datapath Regularity Extraction. In *Proc. 1996 ACM/SIGDA Physical Design Workshop*. 110–117.

NVIDIA. 2023. *CUDA C++ Programming Guide*. Technical Report.

Satoshi Ono and Patrick H Madden. 2005. On Structure And Suboptimality in Placement, In Proceedings of the 2005 Conference on Asia South Pacific Design Automation, ASP-DAC 2005, Shanghai, China, January 18-21, 2005, Ting-Ao Tang (Ed.). *Proceedings of the ASP-DAC 2005. Asia and South Pacific Design Automation Conference, 2005.* 1, 331–336. https://doi.org/10.1109/aspdac.2005.1466184

John K Ousterhout. 1984. Corner Stitching: A Data-Structuring Technique for VLSI Layout Tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 3, 1 (1984), 87–100. https://doi.org/10.1109/tcad.1984.1270061

John K Ousterhout, Gordon T Hamachi, Robert N Mayo, Walter S Scott, and George S Taylor. 1985. The Magic VLSI Layout System. *IEEE Design & Test of Computers* 2, 1 (1985), 19–30. https://doi.org/10.1109/mdt.1985.294681

APV Pais, ML Anido, and CET Oliveira. 2001. Developing a Distributed Architecture for Design Rule Checking. In *Proceedings of the 44th IEEE 2001 Midwest Symposium on Circuits and Systems. MWSCAS 2001 (Cat. No.01CH37257)*, Vol. 2. 678–681. https://doi.org/10.1109/mwscas.2001.986279

Indranil Palit, X Sharon Hu, Joseph Nahas, and Michael Niemier. 2013. Tfet-Based Cellular Neural Network Architectures, In International Symposium on Low Power Electronics and Design (ISLPED), Beijing, China, September 4-6, 2013. *International Symposium on Low Power Electronics and Design*, 236–241. https://doi.org/10.1109/islped.2013.6629301

Yu Pan, Peng Ouyang, Yinglin Zhao, Wang Kang, Shouyi Yin, Youguang Zhang, Weisheng Zhao, and Shaojun Wei. 2018. A Multilevel Cell Stt-Mram-Based Computing In-Memory Accelerator for Binary Convolutional Neural Network. *IEEE Transactions on Magnetics* 54, 11 (7 2018), 1–5. https://doi.org/10.1109/tmag.2018.2848625

Shreepad A Panth, Kambiz Samadi, Yang Du, and Sung Kyu Lim. 2014. Design And Cad Methodologies for Low Power Gate-Level Monolithic 3D Ics, In International Symposium on Low Power Electronics and Design, ISLPED'14, La Jolla, CA, USA - August 11 - 13, 2014, Yuan Xie 0001, Tanay Karnik, Muhammad M. Khellah, and Renu Mehra (Eds.). *International Symposium on Low Power Electronics and Design*, 171–176. https://doi.org/10.1145/2627369.2627642

Ghasem Pasandi, Sreedhar Pratty, David Brown, Yanqing Zhang, Haoxing Ren, and Brucek Khailany. 2021. 2021 ICCAD CAD Contest Problem C: GPU Accelerated Logic Rewriting, In 2021 IEEE/ACM International Conference on Computer Aided Design (ICCAD). *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 1–6. https://doi.org/10.1109/iccad51958.2021.9643521

Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic Differentiation in PyTorch. (10 2017).

Jason Pazis and Ronald Parr. 2011. Generalized Value Functions for Large Action Sets, In Proceedings of the 28th International Conference on Machine Learning, ICML 2011, Bellevue, Washington, USA, June 28 - July 2, 2011, Lise Getoor and Tobias Scheffer (Eds.). *International Conference on Machine Learning*, 1185–1192. https://icml.cc/2011/papers/609_icmlpaper.pdf

Vinicius Possan, Yi-Shan Lu, Alan Mishchenko, Keshav Pingali, Renato Ribas, and Andre Reis. 2018. Unlocking Fine-Grain Parallelism for Aig Rewriting, In 2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), Iris Bahar (Ed.). *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 1–8. https://doi.org/10.1145/3240765.3240861

Angelo PE Rosiello, Fabrizio Ferrandi, Davide Pandini, and Donatella Sciuto. 2007. A Hash-Based Approach for Functional Regularity Extraction During Logic Synthesis, In 2007 IEEE Computer Society Annual Symposium on VLSI (ISVLSI 2007), May 9-11, 2007, Porto Alegre, Brazil. *IEEE Computer Society Annual Symposium on VLSI*, 92–97. https://doi.org/10.1109/isvlsi.2007.5

Nikolay Rubanov. 2006. A High-Performance Subcircuit Recognition Method Based on the Nonlinear Graph Optimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 25, 11 (11 2006), 2353–2363. https://doi.org/10.1109/tcad.2006.881335

Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. 2016. Prioritized Experience Replay, In 4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings, Yoshua Bengio and Yann LeCun (Eds.). *International Conference on Learning Representations* abs/1511.05952. http://arxiv.org/abs/1511.05952

John Schulman, Sergey Levine, Pieter Abbeel, Michael I. Jordan, and Philipp Moritz. 2015. Trust Region Policy Optimization, In Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015, Francis R. Bach and David M. Blei (Eds.). *International Conference on Machine Learning* 37, 1889–1897. http://proceedings.mlr.press/v37/schulman15.html

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal Policy Optimization Algorithms. *ArXiv preprint* abs/1707.06347 (7 2017). https://arxiv.org/abs/1707.06347

Tatjana Serdar and Carl Sechen. 2001. Automatic Datapath Tile Placement And Routing, In DATE. *Proceedings Design, Automation and Test in Europe. Conference and Exhibition 2001*, 552–559. https://doi.org/10.1109/date.2001.915078

Michael Ian Shamos and Dan Hoey. 1976. Geometric Intersection Problems, In 17th Annual Symposium on Foundations of Computer Science (sfcs 1976). *17th Annual Symposium on Foundations of Computer Science (sfcs 1976)*, 208–215. https://doi.org/10.1109/sfcs.1976.16

Yichen Shen, Nicholas C Harris, Scott Skirlo, Mihika Prabhu, Tom Baehr-Jones, Michael Hochberg, Xin Sun, Shijie Zhao, Hugo Larochelle, Dirk Englund, et al. 2017. Deep Learning with Coherent Nanophotonic Circuits. *Nature Photonics* 11, 7 (10 2017), 441. https://doi.org/10.1038/nphoton.2017.93

Leandro Maia Silva, Fabricio Vivas Andrade, Antonio Otavio Fernandes, and Luiz Filipe Menezes Vieira. 2018. Arithmetic Circuit Classification Using Convolutional Neural Networks, In 2018 International Joint Conference on Neural Networks, IJCNN 2018, Rio de Janeiro, Brazil, July 8-13, 2018. *IEEE International Joint Conference on Neural Network*, 1–7. https://doi.org/10.1109/ijcnn.2018.8489382

David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin A. Riedmiller. 2014. Deterministic Policy Gradient Algorithms, In Proceedings of the 31th International Conference on Machine Learning, ICML 2014, Beijing, China, 21-26 June 2014. *International Conference on Machine Learning* 32, 387–395. http://proceedings.mlr.press/v32/silver14.html

David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. 2017. Mastering the Game of Go Without Human Knowledge. *Nature* 550, 7676 (10 2017), 354–359. https://doi.org/10.1038/nature24270

Mathias Soeken, Heinz Riener, Winston Haaswijk, Eleonora Testa, Bruno Schmitt, Giulia Meuli, Fereshte Mozafari, and Giovanni De Micheli. 2018. The Epfl Logic Synthesis Libraries. *ArXiv preprint* abs/1805.05121 (5 2018). https://arxiv.org/abs/1805.05121

Patrick Stotko. 2019. Stdgpu: Efficient Stl-Like Data Structures on the GPU. *ArXiv preprint* abs/1908.05936 (8 2019). https://arxiv.org/abs/1908.05936

Dan Su, Xihong Wu, and Lei Xu. 2010. Gmm-Hmm Acoustic Model Training By a Two Level Procedure with Gaussian Components Determined by Automatic Model Selection, In Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing, ICASSP 2010, 14-19 March 2010, Sheraton Dallas Hotel, Dallas, Texas, USA. *IEEE International Conference on Acoustics, Speech, and Signal Processing*, 4890–4893. https://doi.org/10.1109/icassp.2010.5495122

Pramod Subramanyan, Nestan Tsiskaridze, Wenchao Li, Adria Gascón, Wei Yang Tan, Ashish Tiwari, Natarajan Shankar, Sanjit A Seshia, and Sharad Malik. 2013a. Reverse Engineering Digital Circuits Using Structural And Functional Analyses. *IEEE Transactions on Emerging Topics in Computing* 2, 1 (3 2013), 63–80. https://doi.org/10.1109/tetc.2013.2294918

Pramod Subramanyan, Nestan Tsiskaridze, Kanika Pasricha, Dillon Reisman, Adriana Susnea, and Sharad Malik. 2013b. Reverse Engineering Digital Circuits Using Functional Analysis, In Design, Automation and Test in Europe, DATE 13, Grenoble, France, March 18-22, 2013, Enrico Macii (Ed.). *Design, Automation and Test in Europe*, 1277–1280. https://doi.org/10.7873/date.2013.264

Pramod Subramanyan, Nestan Tsiskaridze, Kanika Pasricha, Dillon Reisman, Adriana Susnea, and Sharad Malik. 2013c. Reverse Engineering Digital Circuits Using Functional Analysis, In 2013 Design, Automation & Test in Europe Conference & Exhibition (DATE), Enrico Macii (Ed.). *Design, Automation and Test in Europe*, 1277–1280. https://doi.org/10.7873/date.2013.264

Qi Sun, Chen Bai, Tinghuan Chen, Hao Geng, Xinyun Zhang, Yang Bai, and Bei Yu. 2021. Fast And Efficient DNN Deployment via Deep Gaussian Transfer Learning, In 2021 IEEE/CVF International Conference on Computer Vision, ICCV 2021, Montreal, QC, Canada, October 10-17, 2021. *IEEE International Conference on Computer Vision*, 5360–5370. https://doi.org/10.1109/iccv48922.2021.00533

Qi Sun, Tinghuan Chen, Jin Miao, and Bei Yu. 2019. Power-Driven DNN Dataflow Optimization on FPGA, In Proceedings of the International Conference on Computer-Aided Design, ICCAD 2019, Westminster, CO, USA, November 4-7, 2019, David Z. Pan (Ed.). *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 1–7. https://doi.org/10.1109/iccad45719.2019.8942085

Xiaoyu Sun, Shihui Yin, Xiaochen Peng, Rui Liu, Jae-sun Seo, and Shimeng Yu. 2018. Xnor-Rram: a Scalable And Parallel Resistive Synaptic Architecture for Binary Neural Networks, In 2018 Design, Automation & Test in Europe Conference & Exhibition (DATE). *Design, Automation and Test in Europe*, 1423–1428. https://doi.org/10.23919/date.2018.8342235

Yihan Sun and Guy E. Blelloch. 2019. Parallel Range, Segment And Rectangle Queries with Augmented Maps, In Proceedings of the Twenty-First Workshop on Algorithm Engineering and Experiments, ALENEX 2019, San Diego, CA, USA, January 7-8, 2019, Stephen G. Kobourov and Henning Meyerhenke (Eds.). *Workshop on Algorithm Engineering and Experimentation*, 159–173. https://doi.org/10.1137/1.9781611975499.13

Richard S Sutton. 1988. Learning to Predict by the Methods of Temporal Differences. *Machine learning* 3, 1 (8 1988), 9–44. https://doi.org/10.1007/bf00115009

Richard S Sutton and Andrew G Barto. 2018. *Reinforcement Learning: An Introduction*. Vol. 16. MIT press. 285–286 pages. https://doi.org/10.1109/tnn.1998.712192

Richard S Sutton, David A McAllester, Satinder P Singh, and Yishay Mansour. 2000. Policy Gradient Methods for Reinforcement Learning with Function Approximation, In Advances in Neural Information Processing Systems 12, [NIPS Conference, Denver, Colorado, USA, November 29 - December 4, 1999], Sara A. Solla, Todd K. Leen, and Klaus-Robert Müller (Eds.). *NIPS*, 1057–1063. http://papers.nips.cc/paper/1713-policy-gradient-methods-for-reinforcement-learning-with-function-approximation

Jakub Szuppe. 2016. Boost.Compute: a Parallel Computing Library for C++ Based on Opencl, In Proceedings of the 4th International Workshop on OpenCL. *International Workshop on OpenCL*, 1–39. https://doi.org/10.1145/2909437.2909454

Taiga Takata and Yusuke Matsunaga. 2009. An Efficient Cut Enumeration for Depth-Optimum Technology Mapping for Lut-Based FPGAs, In Proceedings of the 19th ACM Great Lakes symposium on VLSI, Fabrizio Lombardi, Sanjukta Bhanja, Yehia Massoud, and R. Iris Bahar (Eds.). *ACM Great Lakes Symposium on VLSI*, 351–356. https://doi.org/10.1145/1531542.1531622

Xiaoping Tang and DF Wong. 2001. Fast-Sp: a Fast Algorithm for Block Placement Based on Sequence Pair, In Proceedings of the 2001 Asia and South Pacific design automation conference. *Proceedings of the ASP-DAC 2001. Asia and South Pacific Design Automation Conference 2001 (Cat. No.01EX455)*, 521–526. https://doi.org/10.1145/370155.370523

Mohammad Tehranipoor and Farinaz Koushanfar. 2010. A Survey of Hardware Trojan Taxonomy And Detection. *IEEE Design & Test of Computers* 27, 1 (2010), 10–25. https://doi.org/10.1109/mdt.2010.7

Veronika Thost and Jie Chen. 2021. Directed Acyclic Graph Neural Networks, In 9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021. *International Conference on Learning Representations* abs/2101.07965. https://openreview.net/forum?id=JbuYF437WB6

Jing Tong, Hong Xianlong, Cai Yici, Xu Jingyu, Yang Changqi, Zhang Yiqian, Zhou Qiang, and Wu Weimin. 2002. Challenges to Data-Path Physical Design Inside Soc. *Chinese Journal Of Semiconductors-chinese Edition-* 23, 8 (2002), 785–793.

Wei-Yu Tsai, Xueqing Li, Matthew Jerry, Baihua Xie, Nikhil Shukla, Huichu Liu, Nandhini Chandramoorthy, Matthew Cotter, Arijit Raychowdhury, Donald M Chiarulli, et al. 2016. Enabling New Computation Paradigms with HyperFET-An Emerging Device. *IEEE Transactions on Multi-Scale Computing Systems* 2, 1 (2016), 30–48. https://doi.org/10.1109/tmscs.2016.2519022

Yu-Wen Tsay and Youn-Long Lin. 1995. A Row-Based Cell Placement Method That Utilizes Circuit Structural Properties. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 14, 3 (3 1995), 393–397. https://doi.org/10.1109/43.365130

Kodai Ueyoshi, Kota Ando, Kazutoshi Hirose, Shinya Takamaeda-Yamazaki, Junichiro Kadomoto, Tomoki Miyata, Mototsugu Hamada, Tadahiro Kuroda, and Masato Motomura. 2018. Quest: a 7.49 TOPs Multi-Purpose Log-Quantized DNN Inference Engine Stacked on 96Mb 3D SRAM Using Inductive-Coupling Technology in 40nm CMOS, In 2018 IEEE International Solid-State Circuits Conference, ISSCC 2018, San Francisco, CA, USA, February 11-15, 2018. *IEEE International Solid-State Circuits Conference*, 216–218. https://doi.org/10.1109/isscc.2018.8310261

Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph Attention Networks, In 6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings. *International Conference on Learning Representations* abs/1710.10903. https://doi.org/10.17863/cam.48429

Vasily Volkov. 2010. Better Performance at Lower Occupancy. In *Proceedings of the GPU technology conference, GTC*, Vol. 10. San Jose, CA, 16.

Feng Wang, Guojie Luo, Guangyu Sun, Jiaxi Zhang, Peng Huang, and Jinfeng Kang. 2019b. Parallel Stateful Logic in RRAM: Theoretical Analysis And Arithmetic Design, In 30th IEEE International Conference on Application-specific Systems, Architectures and Processors, ASAP 2019, New York, NY, USA, July 15-17, 2019. *IEEE International Conference on Application-Specific Systems, Architectures, and Processors* 2160, 157–164. https://doi.org/10.1109/asap.2019.000-8

Hanrui Wang, Kuan Wang, Jiacheng Yang, Linxiao Shen, Nan Sun, Hae-Seung Lee, and Song Han. 2020. Gcn-Rl Circuit Designer: Transferable Transistor Sizing with Graph Neural Networks And Reinforcement Learning. *ArXiv preprint* abs/2005.00406 (4 2020), 1–6. https://doi.org/10.1109/dac18072.2020.9218757

Hanrui Wang, Jiacheng Yang, Hae-Seung Lee, and Song Han. 2018. Learning to Design Circuits. *ArXiv preprint* abs/1812.02734 (12 2018). https://arxiv.org/abs/1812.02734

Laung-Terng Wang, Yao-Wen Chang, and Kwang-Ting Tim Cheng. 2009. *Electronic Design Automation: Synthesis, Verification, And Test.* Morgan Kaufmann.

Minjie Wang, Lingfan Yu, Da Zheng, Quan Gan, Yu Gai, Zihao Ye, Mufei Li, Jinjing Zhou, Qi Huang, Chao Ma, et al. 2019c. Deep Graph Library: Towards Efficient And Scalable Deep Learning on Graphs. *ArXiv preprint* abs/1909.01315 (9 2019). https://arxiv.org/abs/1909.01315

Xiao Wang, Houye Ji, Chuan Shi, Bai Wang, Yanfang Ye, Peng Cui, and Philip S. Yu. 2019a. Heterogeneous Graph Attention Network, In The World Wide Web Conference, WWW 2019,

San Francisco, CA, USA, May 13-17, 2019, Ling Liu, Ryen W. White, Amin Mantrach, Fabrizio Silvestri, Julian J. McAuley, Ricardo Baeza-Yates, and Leila Zia (Eds.). *The Web Conference*, 2022–2032. https://doi.org/10.1145/3308558.3313562

Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D Owens. 2016. Gunrock: a High-Performance Graph Processing Library on the GPU, In Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2016, Barcelona, Spain, March 12-16, 2016, Rafael Asenjo 0001 and Tim Harris (Eds.). *ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, 1–12. https://doi.org/10.1145/2851141.2851145

Yu Wang and Hyunchul Shin. 2017. Effective Regularity Extraction And Placement Techniques for Datapath-Intensive Circuits. *IET Circuits Devices Syst.* 11, 5 (5 2017), 512–519. https://doi.org/10.1049/iet-cds.2016.0249

Ziyu Wang, Victor Bapst, Nicolas Heess, Volodymyr Mnih, Rémi Munos, Koray Kavukcuoglu, and Nando de Freitas. 2017. Sample Efficient Actor-Critic with Experience Replay, In 5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings. *International Conference on Learning Representations* abs/1611.01224. https://openreview.net/forum?id=HyM25Mqel

Samuel Ward, Duo Ding, and David Z Pan. 2012a. Pade: a High-Performance Placer with Automatic Datapath Extraction And Evaluation through High-Dimensional Data Learning, In The 49th Annual Design Automation Conference 2012, DAC '12, San Francisco, CA, USA, June 3-7, 2012, Patrick Groeneveld, Donatella Sciuto, and Soha Hassoun (Eds.). *DAC Design Automation Conference 2012*, 756–761. https://doi.org/10.1145/2228360.2228497

Samuel I Ward, Myung-Chul Kim, Natarajan Viswanathan, Zhuo Li, Charles Alpert, Earl E Swartzlander Jr, and David Z Pan. 2012b. Keep It Straight: Teaching Placement How to Better Handle Designs with Datapaths, In International Symposium on Physical Design, ISPD'12, Napa, CA, USA, March 25-28, 2012, Jiang Hu and Cheng-Kok Koh (Eds.). *ACM International Symposium on Physical Design*, 79–86. https://doi.org/10.1145/2160916.2160935

Samuel I Ward, Myung-Chul Kim, Natarajan Viswanathan, Zhuo Li, Charles J Alpert, Earl E Swartzlander, and David Z Pan. 2013. Structure-Aware Placement Techniques for Designs with Datapaths. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 32, 2 (2 2013), 228–241. https://doi.org/10.1109/tcad.2012.2233862

Samuel I Ward, David A Papa, Zhuo Li, Cliff N Sze, Charles J Alpert, and Earl Swartzlander. 2011. Quantifying Academic Placer Performance on Custom Designs, In Proceedings of the 2011 International Symposium on Physical Design, ISPD 2011, Santa Barbara, California, USA, March 27-30, 2011, Yao-Wen Chang and Jiang Hu (Eds.). *ACM International Symposium on Physical Design*, 91–98. https://doi.org/10.1145/1960397.1960420

Christopher JCH Watkins and Peter Dayan. 1992. Q-Learning. *Machine learning* 8, 3-4 (5 1992), 279–292. https://doi.org/10.1007/bf00992698

Xing Wei, Yi Diao, Tak-Kei Lam, and Yu-Liang Wu. 2015. A Universal Macro Block Mapping Scheme for Arithmetic Circuits, In Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition, DATE 2015, Grenoble, France, March 9-13, 2015, Wolfgang Nebel and David Atienza (Eds.). *Design, Automation and Test in Europe*, 1629–1634. https://doi.org/10.7873/date.2015.0746

Xuechao Wei, Cody Hao Yu, Peng Zhang, Youxiang Chen, Yuxin Wang, Han Hu, Yun Liang, and Jason Cong. 2017. Automated Systolic Array Architecture Synthesis for High Throughput CNN Inference on FPGAs, In Proceedings of the 54th Annual Design Automation Conference, DAC 2017, Austin, TX, USA, June 18-22, 2017. *Design Automation Conference*, 1–6. https://doi.org/10.1145/3061639.3062207

Sandra Wienke, Paul Springer, Christian Terboven, and Dieter an Mey. 2012. Openacc—First Experiences with Real-World Applications, In Euro-Par 2012 Parallel Processing: 18th International Conference, Euro-Par 2012, Rhodes Island, Greece, August 27-31, 2012. Proceedings 18, Christos Kaklamanis, Theodore S. Papatheodorou, and Paul G. Spirakis (Eds.). *European Conference on Parallel Processing* 7484, 859–870. https://doi.org/10.1007/978-3-642-32820-6_85

Dan E Willard. 1985. New Data Structures for Orthogonal Range Queries. *SIAM journal on computing (Print)* 14, 1 (2 1985), 232–253. https://doi.org/10.1137/0214019

Ronald J Williams. 1992. Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning. *Machine learning* 8, 3-4 (5 1992), 229–256. https://doi.org/10.1007/978-1-4615-3618-5_2

Michael Wolfe. 2010. Implementing the Pgi Accelerator Model, In Proceedings of the 3rd workshop on general-purpose computation on graphics processing units, David R. Kaeli and Miriam Leeser (Eds.). *GPGPU-3* 425, 43–50. https://doi.org/10.1145/1735688.1735697

D. F. Wong and C. L. Liu. 1986. A New Algorithm for Floorplan Design, In Proceedings of the 23rd ACM/IEEE Design Automation Conference (Las Vegas, Nevada, USA). *23rd ACM/IEEE Design Automation Conference*, 101–107. https://doi.org/10.1145/318013.318030

Hua Xiang, Minsik Cho, Haoxing Ren, Matthew Ziegler, and Ruchir Puri. 2013. Network Flow Based Datapath Bit Slicing, In International Symposium on Physical Design, ISPD'13, Stateline, NV, USA, March 24-27, 2013, Cheng-Kok Koh and Cliff C. N. Sze (Eds.). *ACM International Symposium on Physical Design*, 139–146. https://doi.org/10.1145/2451916.2451954

Zhiyao Xie, Yu-Hung Huang, Guan-Qi Fang, Haoxing Ren, Shao-Yun Fang, Yiran Chen, and Jiang Hu. 2018. Routenet: Routability Prediction for Mixed-Size Designs Using Convolutional Neural Network. In *Proceedings of the International Conference on Computer-Aided Design*, Iris Bahar (Ed.). IEEE, 1–8. https://doi.org/10.1145/3240765.3240843

Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2019. How Powerful Are Graph Neural Networks?, In 7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019. *International Conference on Learning Representations* abs/1810.00826. https://openreview.net/forum?id=ryGs6iA5Km

Hao Yan, Hebin R Cherian, Ethan C Ahn, and Lide Duan. 2018. Celia: a Device And Architecture Co-Design Framework for Stt-Mram-Based Deep Learning Acceleration. In *Proceedings of the 2018 International Conference on Supercomputing*. 149–159. https://doi.org/10.1145/3205289.3205297

Saeyang Yang. 1991. *Logic Synthesis And Optimization Benchmarks User Guide: Version 3.0*. Microelectronics Center of North Carolina (MCNC).

Peng Yao, Huaqiang Wu, Bin Gao, Jianshi Tang, Qingtian Zhang, Wenqiang Zhang, J Joshua Yang, and He Qian. 2020. Fully Hardware-Implemented Memristor Convolutional Neural Network. *Nature* 577, 7792 (1 2020), 641–646. https://doi.org/10.1038/s41586-020-1942-4

Terry Tao Ye, Samit Chaudhuri, Felix Huang, Hamid Savoj, and Giovanni De Micheli. 2002. Physical Synthesis for Asic Datapath Circuits, In ISCAS (4). *IEEE International Symposium on Circuits and Systems proceedings* 3, Iii–iii. https://doi.org/10.1109/iscas.2002.1010236

T Tao Ye and Giovanni De Micheli. 2000. Data Path Placement with Regularity, In Proceedings of the 2000 IEEE/ACM International Conference on Computer-Aided Design, 2000, San Jose, California, USA, November 5-9, 2000, Ellen Sentovich (Ed.). *International Conference on Computer Aided Design*, 264–270. https://doi.org/10.1109/iccad.2000.896484

Emre Yolcu and Barnabás Póczos. 2019. Learning Local Search Heuristics for Boolean Satisfiability, In Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada, Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché Buc, Emily B. Fox, and Roman Garnett (Eds.). *Neural Information Processing Systems*, 7990–8001. https://proceedings.neurips.cc/paper/2019/hash/12e59a33dea1bf0630f46edfe13d6ea2-Abstract.html

Evangeline FY Young, Chris CN Chu, and Zion Cien Shen. 2003. Twin Binary Sequences: a Nonredundant Representation for General Nonslicing Floorplan. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 22, 4 (4 2003), 457–469. https://doi.org/10.1109/tcad.2003.809651

Cunxi Yu and Maciej Ciesielski. 2016. Automatic Word-Level Abstraction of Datapath, In IEEE International Symposium on Circuits and Systems, ISCAS 2016, Montréal, QC, Canada, May 22-25, 2016. *International Symposium on Circuits and Systems*, 1718–1721. https://doi.org/10.1109/iscas.2016.7538899

Tom Zahavy, Matan Haroush, Nadav Merlis, Daniel J. Mankowitz, and Shie Mannor. 2018. Learn What Not to Learn: Action Elimination with Deep Reinforcement Learning, In Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada, Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett (Eds.). *Neural Information Processing Systems*, 3566–3577. https://proceedings.neurips.cc/paper/2018/hash/645098b086d2f9e1e0e939c27f9f2d6f-Abstract.html

Wei Zeng, Azadeh Davoodi, and Rasit Onur Topaloglu. 2020. Explainable DRC Hotspot Prediction with Random Forest And Shap Tree Explainer, In 2020 Design, Automation & Test in Europe Conference & Exhibition, DATE 2020, Grenoble, France, March 9-13, 2020. *Design, Automation and Test in Europe*, 1151–1156. https://doi.org/10.23919/date48585.2020.9116488

Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. 2015. Optimizing FPGA-Based Accelerator Design for Deep Convolutional Neural Networks, In Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, CA, USA, February 22-24, 2015, George A. Constantinides and Deming Chen (Eds.). *Symposium on Field Programmable Gate Arrays*, 161–170. https://doi.org/10.1145/2684746.2689060

Chuxu Zhang, Dongjin Song, Chao Huang, Ananthram Swami, and Nitesh V. Chawla. 2019b. Heterogeneous Graph Neural Network, In Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2019, Anchorage, AK, USA, August 4-8, 2019, Ankur Teredesai, Vipin Kumar, Ying Li, Rómer Rosales, Evimaria Terzi, and George Karypis (Eds.). *Knowledge Discovery and Data Mining*, 793–803. https://doi.org/10.1145/3292500.3330961

Jiaxi Zhang, Wentai Zhang, Guojie Luo, Xuechao Wei, Yun Liang, and Jason Cong. 2019c. Frequency Improvement of Systolic Array-Based CNNs on FPGAs, In IEEE International Symposium on Circuits and Systems, ISCAS 2019, Sapporo, Japan, May 26-29, 2019. *International Symposium on Circuits and Systems*, 1–4. https://doi.org/10.1109/iscas.2019.8702071

Muhan Zhang, Shali Jiang, Zhicheng Cui, Roman Garnett, and Yixin Chen. 2019a. D-Vae: a Variational Autoencoder for Directed Acyclic Graphs, In Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada, Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché Buc, Emily B. Fox, and Roman Garnett (Eds.). *Neural Information Processing Systems*, 1586–1598. https://proceedings.neurips.cc/paper/2019/hash/e205ee2a5de471a70c1fd1b46033a75f-Abstract.html

Xiaofan Zhang, Junsong Wang, Chao Zhu, Yonghua Lin, Jinjun Xiong, Wen-mei Hwu, and Deming Chen. 2018. DNNbuilder: An Automated Tool for Building High-Performance DNN Hardware Accelerators for FPGAs. In *Proceedings of the International Conference on Computer-Aided Design*, Iris Bahar (Ed.). 1–8. https://doi.org/10.1145/3240765.3240801

Yanqing Zhang, Haoxing Ren, Ben Keller, and Brucek Khailany. 2020. Problem C: GPU Accelerated Logic Re-Simulation. In *Proceedings of the 39th International Conference on Computer-Aided Design*. 1–4. https://doi.org/10.1145/3400302.3415740

Yanqing Zhang, Haoxing Ren, Akshay Sridharan, and Brucek Khailany. 2022. Gatspi: GPU Accelerated Gate-Level Simulation for Power Improvement. *ArXiv preprint* abs/2203.06117 (3 2022), 1231–1236. https://doi.org/10.1145/3489517.3530601

Wenqian Zhao, Qi Sun, Yang Bai, Wenbo Li, Haisheng Zheng, Bei Yu, and Martin DF Wong. 2021. A High-Performance Accelerator for Super-Resolution Processing on Embedded GPU, In 2021 IEEE/ACM International Conference on Computer Aided Design (ICCAD). *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)* abs/2303.08999, 1–9. https://doi.org/10.1109/tcad.2023.3241110

Yangming Zhou, Jin-Kao Hao, and Béatrice Duval. 2016. Reinforcement Learning Based Local Search for Grouping Problems: a Case Study on Graph Coloring. *Expert Systems with Applications* 64 (4 2016), 412–422. https://doi.org/10.1016/j.eswa.2016.07.047

Binwu Zhu, Xinyun Zhang, Yibo Lin, Bei Yu, and Martin Wong. 2022. Efficient Design Rule Checking Script Generation via Key Information Extraction, In MLCAD '22: 2022 ACM/IEEE Workshop on Machine Learning for CAD, Virtual Event China, September 12 - 13, 2022. *Workshop on Machine Learning for CAD*, 77–82. https://doi.org/10.1145/3551901.3556494

Brian D Ziebart. 2010. Modeling Purposeful Adaptive Behavior with the Principle of Maximum Causal Entropy. (2010). https://doi.org/10.1184/r1/6720692.v1

Ying Zuo, Bohan Li, Yujun Zhao, Yue Jiang, You-Chiuan Chen, Peng Chen, Gyu-Boong Jo, Junwei Liu, and Shengwang Du. 2019. All-Optical Neural Network with Nonlinear Activation Functions. *Optica* 6, 9 (4 2019), 1132–1137. https://doi.org/10.1364/optica.6.001132