UNIVERSITY OF CALIFORNIA SAN DIEGO

# AI for Design Optimization and Design for AI Acceleration

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

in

Electrical Engineering (Computer Engineering)

by

Uday Mallappa

Committee in charge:

Professor Chung-Kuan Cheng, Chair
Professor Bill Lin, Co-Chair
Professor Tajana Šimunić Rosing, Co-Chair
Professor Sicun Gao
Professor Rajesh K. Gupta

2022

## DEDICATION

*To my mother, Renuka (Manu) Devi*

*and my father, Jayantha Sharma*

TABLE OF CONTENTS

Dissertation Approval Page . . . iii

Dedication . . . iv

Table of Contents . . . v

List of Figures . . . viii

List of Tables . . . xiii

Acknowledgements . . . xv

Vita . . . xviii

Abstract of the Dissertation . . . xx

Chapter 1 Introduction . . . 1
1.1 Machine Learning for Physical Design . . . 2
1.1.1 Estimators for Timing Outcomes . . . 3
1.1.2 Active Learning for Improved Placement Quality . . . 3
1.2 Boolean Satisfiability as a Function Fitting Problem . . . 6
1.3 Domain Specialized Hardware . . . 8
1.3.1 Machine Learning Acceleration . . . 8
1.4 Thesis Outline . . . 10

Chapter 2 AI for EDA: Design Analysis . . . 12
2.1 Introduction . . . 12
2.2 Related Work . . . 16
2.3 PBA-GBA: Preliminaries . . . 17
2.3.1 Intuition for Bigram-Based Modeling . . . 18
2.3.2 Selection of Features . . . 19
2.3.3 Classification and regression trees . . . 21
2.4 PBA-GBA: Modeling Methodology . . . 23
2.4.1 Reporting Metrics . . . 23
2.4.2 Model Definition . . . 25
2.4.3 Modeling Flow . . . 26
2.5 PBA-GBA: Experimental Validation . . . 27
2.5.1 Design of Experiments . . . 27
2.5.2 Results . . . 28
2.6 Unobserved Corner Prediction: Preliminaries . . . 30
2.6.1 Intuition for Multivariate Linear Regression . . . 32
2.7 Unobserved Corner Prediction: Modeling Methodology . . . 33
2.7.1 Data Generation . . . 35

2.8 Unobserved Corner Prediction: Experimental Validation . . . . . 38
2.8.1 Reporting Metrics . . . . . 39
2.8.2 Design of Experiments . . . . . 40
2.8.3 Experimental Results . . . . . 43
2.9 Conclusion . . . . . 45

Chapter 3 AI for EDA: Design Optimization . . . . . 49
3.1 Introduction . . . . . 50
3.2 Related Work . . . . . 51
3.3 Placement as MDP . . . . . 52
3.3.1 Markov Decision Process (MDP) . . . . . 52
3.3.2 Reinforcement Learning . . . . . 52
3.3.3 Deep Q Learning . . . . . 54
3.3.4 Satisfiability Modulo Theories (SMT) . . . . . 54
3.4 RL for Detailed Placement . . . . . 55
3.4.1 Stage 1 (coarse-grained placement) . . . . . 55
3.4.2 Stage 2 (fine-grained placement) . . . . . 57
3.5 Experimental Setup . . . . . 59
3.6 Experiments . . . . . 61
3.6.1 Improvement over Global-Placement . . . . . 61
3.6.2 Improvement over Commercial Detailed-Placement Flow . . . . . 63
3.7 Conclusion . . . . . 64

Chapter 4 AI for EDA: Network-on-Chip Routing . . . . . 67
4.1 Introduction . . . . . 67
4.2 Related Work . . . . . 70
4.3 Joint Application-Aware Oblivious Routing and Static VC Assignment (`JARVA`) 71
4.3.1 Network Model . . . . . 71
4.3.2 Deadlock Avoidance in a 2D Mesh in Wormhole Routing . . . . . 72
4.3.3 Deadlock Avoidance in a 2D Mesh with Static VC Assignment . . . . . 74
4.3.4 SMT Formulation . . . . . 76
4.3.5 Router Implementation . . . . . 81
4.4 Evaluation . . . . . 81
4.4.1 Experimental Setup . . . . . 81
4.4.2 Synthetic Traffic . . . . . 82
4.4.3 Camcorder Scenarios . . . . . 84
4.4.4 Detailed Flit-Level Simulations . . . . . 84
4.5 Conclusion . . . . . 89

Chapter 5 AI Acceleration: Skipping Ineffectual Computations . . . . . 90
5.1 Introduction . . . . . 90
5.2 Ineffectual Output Skipping . . . . . 93
5.2.1 Inference Framework . . . . . 94
5.2.2 Model Calibration . . . . . 97

5.2.3 Hyperparameters Exploration . . . . . 99
5.3 TermiNETor Architecture . . . . . 101
5.3.1 Overview . . . . . 101
5.3.2 Dataflow . . . . . 102
5.3.3 Data Reuse . . . . . 105
5.3.4 Processing Elements . . . . . 106
5.4 Experimental Results . . . . . 107
5.4.1 General Setup . . . . . 107
5.4.2 Operation Reduction . . . . . 108
5.4.3 TermiNETor Accelerator Evaluation . . . . . 111
5.4.4 Comparison with Previous Work . . . . . 113
5.5 Conclusion . . . . . 114
5.5.1 Acknowledgements . . . . . 115

Chapter 6 AI Acceleration: Patterned DNN . . . . . 116
6.1 Introduction . . . . . 116
6.2 Patterned Neural Network . . . . . 119
6.2.1 Motivation . . . . . 119
6.2.2 Pattern Selection . . . . . 120
6.2.3 Free Filters . . . . . 121
6.2.4 Patterned Model Training . . . . . 123
6.3 PatterNet Architecture . . . . . 124
6.3.1 Overview . . . . . 124
6.3.2 Data Flow . . . . . 125
6.3.3 Data Reuse . . . . . 126
6.3.4 Processing Units . . . . . 127
6.4 Experiments and Results . . . . . 128
6.4.1 Experimental Setup . . . . . 128
6.4.2 Operation and Memory Reduction . . . . . 129
6.4.3 PatterNet Accelerator Details . . . . . 130
6.4.4 PatterNet Scalability . . . . . 131
6.4.5 Comparison with Previous Work . . . . . 132
6.4.6 Physical Design and Tapeout . . . . . 133
6.5 Conclusion . . . . . 134
6.5.1 Acknowledgements . . . . . 135

Chapter 7 Conclusions . . . . . 136
7.1 Thesis Summary . . . . . 136
7.2 Future Directions . . . . . 138
7.2.1 AI for Design Optimization . . . . . 138
7.2.2 ASIC Design for AI Acceleration . . . . . 139

Bibliography . . . . . 140

LIST OF FIGURES

Figure 1.1. A major portion of the IC design cost is invested in the software tools that automate the design process [1]. Also, the software cost for a 5*nm* is nearly 100% larger than a 7*nm* chip. . . . . . . . . . . 1

Figure 1.2. The reduction of minimum metal pitch (MMP) in recent years, indicates slowing of the regular doubling of integration density (Moore's law). . . . . 2

Figure 1.3. Relationship between the SAT problem and binary neural networks. . . . . . 6

Figure 1.4. AI acceleration startups [2] with more than \$50M funding as of November 2022. . . . . . . . . . . 8

Figure 1.5. The Roofline model for TPU ASIC (blue) has much longer slanted line as compared to NVIDIA K80 GPU (red) and Intel Haswell CPU (yellow) [3]. 9

Figure 1.6. Dependencies between AI, EDA and ASIC Design. Inspired from Roger Penrose's metaphysical framework [4]. . . . . . . . . . 10

Figure 2.1. Transition propagation in (a) GBA mode and (b) PBA mode. . . . . . . . . . . 13

Figure 2.2. Ratio of PBA runtime to GBA runtime on log scale (commercial signoff STA tool; 28nm FDSOI foundry enablement) for public-domain design examples (see Table 2.5 for details) ranging in size from 530 flip-flops and 11K signal nets to 350K flip-flops and 960K signal nets. . . . . . . . . . 14

Figure 2.3. PBA-GBA divergence for the megaboom design (350K flip-flops, 960K signal nets) signed off at 1.2ns clock period in 28nm FDSOI technology. . 15

Figure 2.4. PBA-GBA divergence for a design signed off at 1.2ns ( 990K standard cells) in 28nm FDSOI technology. Shown: timing analysis for the same path in (a) GBA mode and (b) PBA mode. . . . . . . . . . 17

Figure 2.5. Bigram-based model of a timing path. The timing path is represented as a series of four bigram units. . . . . . . . . . 19

Figure 2.6. The impact of (a) Dropping any single one of the 13 parameters (indexed as above); the peak loss of accuracy corresponds to $TR\_ret_{gba}$. (b) Dropping any pair of the 13 parameters at a time; the x-axis gives, from left to right, C(13,2) = 78 pairs (1,2), (1,3), ..., (12,13). . . . . . . . . . 21

Figure 2.7. Tree-based classification with 1.7M training samples and 13 parameters. Parameter X[4], which corresponds to $TR\_rat_{gba}$, splits the data space into 75% and 25% with a split value of 0.493, indicating its importance in classifying input data. . . . . . . . . . 24

Figure 2.8. PBA versus GBA path-consistent arrival time plots using a commercial timer in 28nm FDSOI technology. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25

Figure 2.9. Our modeling flow. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 26

Figure 2.10. Results of Experiment 1 for actual GBA path arrival time (top row) and predicted PBA path arrival time (bottom row) versus actual PBA path arrival time for megaboom (left), leon3mp (middle) and netcard (right). . . 29

Figure 2.11. Results of Experiment 2 for actual GBA path arrival time (top row) and predicted PBA path arrival time (bottom row) versus actual PBA path arrival time for megaboom (left), leon3mp (middle) and netcard (right). . . 30

Figure 2.12. Results of Experiment 3 for actual GBA path arrival time (top row) and predicted PBA path arrival time (bottom row) versus actual PBA path arrival time for megaboom (left), leon3mp (middle) and netcard (right). . . 30

Figure 2.13. Visualization of timing prediction as a problem in matrix completion. . . . . 32

Figure 2.14. Our modeling flow. The potential feedback loop (blue arrows) is discussed in Sections VI below. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35

Figure 2.15. Illustration of artificial circuits. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38

Figure 2.16. Relative variance captured by successive principal components of the first four data sets in Table 2.10 (44 analysis corners). In each data set, the variance is concentrated in a subspace of much lower dimensionality than the total number of analysis corners. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41

Figure 2.17. Results of Experiments 1, 2, 3, 4 and 5 (top to bottom): Plots of $\varepsilon_{mse}$ (red), $\varepsilon_{99p}$ (blue) and $\varepsilon_{99p99}$ (green) versus $n$, for designs *dec_viterbi*, *netcard*, *leon3mp* and *megaboom* (left to right). . . . . . . . . . . . . . . . . . . . . . . . . . . . 46

Figure 2.18. Results of Experiment 6 using sub-14nm technology libraries. Plots of $\varepsilon_{mse}$ (red), $\varepsilon_{99p}$ (blue) and $\varepsilon_{99p99}$ (green) versus $n$ for industrial designs (a) *prod1* (b) *prod2* and (c) *prod3*. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 47

Figure 3.1. The problem instances of `RLPlace` , shown on a blurred layout of our design. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 51

Figure 3.2. Neural network takes state of the window as the input and learns to distinguish good actions (grid swaps) against the bad actions. . . . . . . . . . . . . . . 54

Figure 3.3. The flow on the left shows the learning phase of our RL framework and the right is model deployment. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 55

Figure 3.4. The components of MDP (left). As the window is slided across the layout, RL agent finds the optimal grid arrangement in each window. . . . . . . . . . . 56

Figure 3.5. Stage 2 involves re-arrangement of cells within grids of the window. . . . . . 58

Figure 3.6. Optimal re-arrangement problem is analogous to the Rubik's cube problem, with an added complexity of net connectivity between the miniature cubes. 60

Figure 3.7. The MSE of the policy network (left), reward improvement as a function of training episodes (middle) and the final episode's cumulative reward (right). 61

Figure 3.8. HPWL improvement of `RLPlace` , with global-placement of *Multiplier Unit 1* as the start point. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 62

Figure 3.9. HPWL improvement during the first 4K iterations (for four variants of *design2*). . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 63

Figure 3.10. Plots showing substantial STWL improvement (over 35%) for a variant of *design1*, when run for sufficiently longer number of iterations (we run for 100K iterations for this variant). . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 64

Figure 3.11. Plots showing HPWL improvement over highly-optimized design variants of our IPs . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 65

Figure 3.12. Overlap-free layout (blurred) of our designs, generated from the RLPlace algorithm. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 65

Figure 3.13. Plots showing the benefit of SMT/SAT based cell distribution with greedy-swap (Stage 2), and comparison of RL-based heuristic with random heuristics (Stage 1). . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 66

Figure 4.1. The turns (clockwise and counter-clockwise) that are allowed (solid lines) and prohibited (dashed lines) under (a) the West-First turn model, (b) the North-Last turn model, and (c) the Negative-First turn model, each with four rotations (steps of 90 degrees), for a total of 12 different turn models [5]. 72

Figure 4.2. Synthetic traffic patterns. See Table 4.3 for descriptions. . . . . . . . . . . . . . . 83

Figure 4.3. Summary of heterogeneous cores arranged in different 2D network configurations for a Camcorder application. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 85

Figure 4.4. Latency as a function of injection rate for various routing algorithms on (a) Transpose (b) Shuffle (c) BitComp (d) Hotspot1 (e) Hotspot2 and (f) Hotspot3 patterns. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 87

Figure 4.5. Latency as a function of injection rate for various routing algorithms on the camcorder(a-d) applications. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 88

Figure 5.1. Ineffectual outputs due to ReLU and max pooling. . . . . . . . . . . . . . . . . . . . . 91

Figure 5.2. Ineffectual outputs of VGG16 on CIFAR-100 images. . . . . . . . . . . . . . . . . . 94

Figure 5.3. Early output prediction of `TermiNETor`. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 96

Figure 5.4. An example of negative-to-positive sign flip. . . . . . . . . . . . . . . . . . . . . . . . . . 97

Figure 5.5. Percentage of false negative for two layers of a pretrained VGG16 model on CIFAR-100 dataset. The X-axis is the number of used weight bits. . . . . 99

Figure 5.6. Weight histograms before model calibration (left) and after (right) for two layers of VGG16. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 100

Figure 5.7. Overview of `TermiNETor`'s baseline architecture. . . . . . . . . . . . . . . . . . . . . . 102

Figure 5.8. A sample input and filters for `TermiNETor` dataflow explanation. . . . . . . . . 104

Figure 5.9. A processing element (PE) of `TermiNETor`. . . . . . . . . . . . . . . . . . . . . . . . . . 106

Figure 5.10. Physical layout of `TermiNETor` accelerator, implemented in TSMC 40 nm technology. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 108

Figure 5.11. Scalability of performance, area, power, and utilization of `TermiNETor` running different image sizes.. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 112

Figure 5.12. Energy per classification, normalized to GPU. . . . . . . . . . . . . . . . . . . . . . . . 113

Figure 6.1. Weight clustering with (a) computation reuse and (b) cluster reuse between filters. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 118

Figure 6.2. Convolution operation. Filters $f_1$ and $f_2$ use the same pattern with different unique weights. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 119

Figure 6.3. Memory and operation saving of patterned weight clustering when $N$ filters share one pattern. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 120

Figure 6.4. Selecting and sharing weight patterns. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 122

Figure 6.5. PatterNet (1) architecture, (2) processing element, and (3) data flow. . . . . . 123

Figure 6.6. Scalability of performance, power, and energy of PatterNet running VGG16 with (a) 224×224 (b) 32×32 images. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 132

Figure 6.7. Comparison of PatterNet, SCNN [6], and FuseKNA [7] energy efficiency normalized to GPU. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 133

Figure 6.8. Physical layout of `PatterNet` accelerator; wirebond design with 145 IO pads and a total silicon area of $11.25mm^2$ in TSMC40nm. . . . . . . . . . . . . . . . 134

LIST OF TABLES

Table 2.1. Terms and definitions for PBA-GBA. 18

Table 2.2. Regression versus classification trees (netcard). 23

Table 2.3. PBA-GBA divergence metric. 24

Table 2.4. Model prediction-based divergence metrics. 25

Table 2.5. Design data used for experiments. 27

Table 2.6. Model divergence improvement in Experiment 1. 28

Table 2.7. Model divergence improvement in Experiment 2. 29

Table 2.8. Model divergence improvement in Experiment 3. 29

Table 2.9. Terms and definitions for Unobserved Corner Prediction. 31

Table 2.10. Design data used for experiments. 36

Table 2.11. Space of 28nm FDSOI analysis corners (Expts 1-4). 36

Table 2.12. Model accuracy metrics. 42

Table 3.1. Wirelength (um) comparison of RLPlace with a commercial tool; optimized output from global placement (Baseline) as the start point for both. 62

Table 4.1. Terms and Definitions. 76

Table 4.2. `JARVA` 's formulation complexity. 80

Table 4.3. Synthetic Traffic Patterns 82

Table 4.4. Comparison of Maximum Channel Load on Synthetic Benchmarks. 83

Table 4.5. MCL Comparison on Synthetic Benchmarks. 83

Table 4.6. Comparison of Maximum Channel Load on a Camcorder Application. 85

Table 5.1. Accuracy and operation comparison of `TermiNETor` with baselines. 107

Table 5.2. Operation reduction with $\sim$2% accuracy budget. 110

Table 5.3. Characterization of `TermiNETor` components. 110

Table 6.1. Comparing PatterNet with baseline and Hrank [8]. . . . . . . . . . . . . . . . . . . . . . 130

Table 6.2. Memory size of baseline PatterNet architecture. . . . . . . . . . . . . . . . . . . . . . . . 130

Table 6.3. Characterization of PatterNet components. . . . . . . . . . . . . . . . . . . . . . . . . . . . 131

Table 6.4. Metrics of 8×16 array for large and small images. . . . . . . . . . . . . . . . . . . . . 132

## ACKNOWLEDGEMENTS

Foremost, I would like to thank my advisor, Professor Chung-Kuan Cheng, for providing me with a free hand in exploring various research problems. His compassionate belief system ensures that his students get to work on problems that interest them, making it a satisfying research experience. I would like to thank my co-advisor, Professor Tajana Rosing (along with TSMC), for providing me with a rewarding experience of silicon implementation for our novel algorithmic idea. I would also like to thank my other co-advisor, Professor Bill Lin, for introducing me to interesting optimization problems in network-based communication systems. Both Professor Rosing and Professor Lin have supported me immensely, during my journey. I would like to thank Professor Andrew Kahng, for bringing me to UC San Diego and offering his guidance in conducting great research. Among many one-liners from him, one advice that helped me remarkably was “If it is not written, it does not exist”.

In addition to my advisors, I would like to thank Professor Lawrence Saul for guiding me on multiple research problems. I would also like to thank my thesis committee members; Professor Rajesh Gupta and Professor Sicun Gao for taking the time to review my thesis and offering valuable comments during my University Qualifying Exam and afterwards. Special thanks to Professor Sicun Gao, for his guidance on various interesting problems in logic synthesis, reinforcement learning, and Boolean optimization. I thank Dr. Siddhartha Nath (Intel), Dr. Tuck-Boon Chan (Google), Dr. Jiajia Li (Apple), Professor Sachin Sapatnekar (Univesity of Minnesota), David Brown (Nvidia), Sreedhar Pratty (Nvidia), Dr. Somdeb Majumdar (Intel Labs), and Dr. Mariano Phielipp (Intel Labs) for their mentorship on multiple projects.

I would like to thank all my colleagues in UCSD VLSI CAD Laboratory (Kwangsoo Han, Hyein Lee, Minsoo Kim, Lutong Wang, Bangqi Xu, Mateus Fogaça, Mingyu Woo, Hsin Yu Liu, Payal Agarwal, Siddharth Singh, and Shangyuan Tong), my colleagues in VLSI Lab (Chester Holtz, Yucheng Wang, Daeyeal Lee, Dongwon Park, and Chia-Tung Ho), my friends in SEELab (Haichao Yang, Monil Shah, and Pranav Gangwar), and the OpenROAD team for multiple research collaborations. I would like to specially thank Behnam Khaleghi for stimulating discussions and great memories during the last two years of my Ph.D.

Though my mother, Renuka, is not physically present to share this moment with me, I would like to emphasize her influence on my life. She had an incredible persona of accepting adversities in her stride and has always embraced criticism. These qualities turned out to be distinctly helpful during the challenging times of my Ph.D. journey, and I hope to hang on to these qualities for the rest of my life. Also, I am eternally indebted to my father, who inspired me to be original in my thoughts and motivated me to look beyond short-term monetary rewards. The faith and support from my parents encouraged me to seek opportunities that genuinely interest me. I am grateful to my uncle, Mr. M. Srikanth, who has been a strong tower of strength throughout my career, and life would have been unusually different without his support and mentorship. I would love to thank my wife, Anagha, for supporting me in many ways, and bearing with me. Lastly, I thank my brother Eshwar for his constant support, my sister, and the rest of my family for their faith.

Chapter 2 contains material from "Using machine learning to predict path-based slack from graph-based timing analysis", by Andrew B. Kahng, Uday Mallappa, and Lawrence Saul, which appears in International Conference on Computer Design (ICCD), October 2018; "Unobserved Corner Prediction: Reducing Timing Analysis Effort for Faster Design Convergence in Advanced-Node Design", by Andrew B. Kahng, Uday Mallappa, Lawrence Saul and Shangyuan Tong, which appears in Design, Automation and Test in Europe Conference & Exhibition (DATE), March 2019. The dissertation author was the primary investigator and author of both these papers.

Chapter 3 contains materials from "RLPlace: deep RL guided heuristics for detailed placement optimization", by Uday Mallappa, Sreedhar Pratty, and David Brown, which appears in Design, Automation and Test in Europe Conference & Exhibition (DATE), March 2022. The dissertation author was the primary investigator and author of this paper.

Chapter 4 contains materials from "Joint Application-Aware Oblivious Routing and Static Virtual Channel Allocation", by Uday Mallappa, Chung-Kuan Cheng, and Bill Lin, which appears in IEEE Design and Test, Aug 2022. The dissertation author was the primary investigator and author of this paper.

Chapter 5 contains materials from “TermiNETor: Early Convolution Termination for Efficient Deep Neural Networks”, by Uday Mallappa, Pranav Gangwar, Behnam Khaleghi, Haichao Yang, and Tajana Rosing, which appears in International Conference on Computer Design (ICCD), October 2022. The dissertation author was the primary investigator and author of this paper.

Chapter 6 contains materials from “PatterNet: Explore and Exploit Filter Patterns for Efficient Deep Neural Networks”, by Uday Mallappa*, Behnam Khaleghi*, Duygu Yaldiz, Haichao Yang, Monil Shah, Jaeyoung Kang, and Tajana Rosing, which appears in Design Automation Conference (DAC), July 2022. The dissertation author was one of the primary investigators and authors of this paper.

My co-authors (David Brown, Prof. Chung-Kuan Cheng, Pranav Gangwar, Prof. Andrew B. Kahng, Jaeyoung Kang, Behnam Khaleghi, Prof. Bill Lin, Sreedhar Pratty, Prof. Tajana Rosing, Prof. Lawrence Saul, Monil Shah, Shangyuan Tong, Duygyu Yaldiz, and Haichao Yang, listed in alphabetical order) have all kindly approved the inclusion of the aforementioned publications in my dissertation.

## VITA

| | |
|---|---|
| 2011 | B.E. (Hons), in Electrical and Electronics Engineering & M.Sc. in Physics, Birla Institute of Technology and Science, India |
| 2011-2012 | Engineer, Ansys, Bangalore, India |
| 2012-2016 | Senior Design Engineer, Qualcomm India Private Limited, Bangalore, India |
| 2016-2017 | Component Design Engineer, Intel India Private Limited, Bangalore, India |
| 2021 | M.S. in Electrical Engineering (Computer Engineering), University of California San Diego |
| 2021 | C.Phil. in Electrical Engineering (Computer Engineering), University of California San Diego |
| 2022 | Ph.D. in Electrical Engineering (Computer Engineering), University of California San Diego |

## PUBLICATIONS

Uday Mallappa, Pranav Gangwar, Behnam Khaleghi, Haichao Yang, and Tajana Rosing, "TerminETor: Early Convolution Termination for Efficient Deep Neural Networks", *International Conference on Computer Design (ICCD)*, 2022.

Uday Mallappa, Chung-Kuan Cheng, and Bill Lin, "JARVA: Joint Application-Aware Oblivious Routing and Static Virtual Channel Allocation", *IEEE Design & Test*, 2022.

Uday Mallappa*, Behnam Khaleghi*, Duygu Yaldiz, Haichao Yang, Monil Shah, Jaeyoung Kang, and Tajana Rosing, "PatterNet: Explore and Exploit Filter Patterns for Efficient Deep Neural Networks", *Design Automation Conference (DAC)*, 2022. *Contributed Equally

Uday Mallappa, Chung-Kuan Cheng, and Bill Lin, "Joint Application-aware oblivious-Routing and Static VC Allocation (JARVA)", Embedded Systems Letters (ESL), 2022.

Uday Mallappa, Sreedhar Pratty, and David Brown, "RLPlace: Deep RL Guided Heuristics for Detailed Placement Optimization", *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2022.

Uday Mallappa and Chung-Kuan Cheng, "GRA-LPO: Graph Convolution Based Leakage PowerOptimization", IEEE/ACM Asia and South Pacific Design Automation Conference (ASP-

DAC), 2021.

Vidya Chabbria, Andrew B. Kahng, Minsoo Kim, Uday Mallappa, Sachin Sapatnekar, and Bangqi Xu “Template-based PDN Synthesis in Floorplan and Placement Using Classifier and CNN Techniques”, IEEE/ACM Asia and South Pacific Design Automation Conference (ASP-DAC), 2020. (Alphabetical Order)

Tutu Ajayi, Vidya A. Chhabria, Mateus Fogaca, 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, Luton Wang, Zhehong Wang, Mingyu Woo, and Bangqi Xu, “Toward an Open-Source Digital Flow: First Learnings from the OpenROAD Project”, IEEE Design Automation Conference (DAC), 2019. (Alphabetical Order)

Andrew B. Kahng, Uday Mallappa, Lawrence Saul, and Shangyuan Tong, “Unobserved Corner Prediction: Reducing Timing Analysis Effort for Faster Design Convergence in Advanced-Node Design”, *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2019. (nominated for Best Paper award, Alphabetical Order)

Andrew B. Kahng, Uday Mallappa, and Lawrence Saul, “Using Machine Learning to Predict Path-Based Slack from Graph-Based Timing Analysis”, *International Conference on Computer Design (ICCD)*, 2018. (Alphabetical Order).

ABSTRACT OF THE DISSERTATION

# AI for Design Optimization and Design for AI Acceleration

by

Uday Mallappa

Doctor of Philosophy
in Electrical Engineering (Computer Engineering)

University of California San Diego, 2022

Professor Chung-Kuan Cheng, Chair
Professor Bill Lin, Co-Chair
Professor Tajana Šimunić Rosing, Co-Chair

Integrated circuit (IC) design at the scale of billions of circuit elements would be unimaginable without the software and services from the Electronic Design Automation (EDA) industry. However, today, the designers using these EDA tools and flows are confronted by long runtimes, high design costs and low power, performance, and area (PPA) gains when transitioning to the latest process nodes. The long tool-runtimes and high tool-license costs make it prohibitively expensive for a thorough design-space exploration. Furthermore, pessimistic margins introduced at various stages of the EDA flow, to balance the accuracy-runtime tradeoff, can result in sub-optimal design implementations. To counter these issues and keep up with the pace of PPA expectations from the market, the dissertation contributes to two promising opportunities at the top of the compute stack: (1) algorithmic improvements and (2) domain-specialized hardware.

For algorithmic contributions, we exploit AI-based techniques (i) to reduce the design and schedule costs of advanced node IC design, and (ii) to efficiently search for optimal design implementations. A significant portion of the design cycle is spent on the static timing analysis (STA) at multiple corners and multiple modes (MCMM). To address the schedule costs of STA engines, we propose a learning model to accurately predict expensive path-based analysis (PBA) results from pessimistic graph-based analysis (GBA). We also devise a MCMM timing model using learning-based techniques, to predict accurate timing results at unobserved signoff corners, using timing results from a small subset of corners. Our PBA-GBA model reduces the maximum PBA-GBA divergence from 50.78ps to 39.46ps, for a 350K-instance design in 28nm FDSOI foundry. Our MCMM timing prediction model uses timing results from 10 observed corners, to predict timing results at the remaining 48 unobserved corners with less than 0.5% relative root mean squared error (RMSE), for a 1M-instance design in 16nm enablement. Besides STA, the two most important and critical phases of the IC design cycle are the placement of standard cells, and the routing tasks at various abstraction levels. To demonstrate the use of learning-based models for efficient search of optimal placement implementation, we propose a reinforcement learning (RL)-based framework `RLPlace` for the task of detailed placement optimization. With the global placement output of two critical IPs as the start point, `RLPlace` achieves up to 1.35% half-perimeter wirelength (HPWL) improvement as compared to the commercial tool's detailed placement results. To efficiently search for optimal routing solutions in network-based communication systems, we propose a SMT-based framework to jointly determine routing and virtual channel (VC) assignment solutions in network-on-chip (NOC) design. Our novel formulation enables better deadlock-free performance, achieving up to 30% better performance than the state-of-the-art application-aware oblivious routing algorithms.

To keep up with the PPA expectations as we navigate towards the post-Moore world, we also propose two novel hardware accelerators for image classification tasks, that exemplify the performance and energy benefits of domain-specialized hardware. To alleviate the computation and energy burden of neural network inference, we focus on two key areas: (i) skipping unnecessary computations, and (i) maximizing the reuse of redundant computations. Our `TermiNETor` framework skips ineffectual computations during the inference of image classification tasks. `TermiNETor` relies on bit-serial weight processing, to dynamically predict and skip the computations that are unnecessary for downstream computations. The `TermiNETor` framework achieves up to 1.7× reduction of operation count compared to non-skipping baseline without accuracy degradation, and the hardware implementation of `TermiNETor` framework improves the average energy efficiency by 3.84× over SCNN [6], and by 1.98× over FuseKNA [7]. Our second accelerator `PatterNet` demonstrates the performance and energy benefits of reusing redundant computations during the inference phase of image classification. `PatterNet` is based on patterned neural networks for computation reuse, and supported with a novel pattern-stationary architecture. With similar accuracy results, our `PatterNet` accelerator reduces the memory and operation count up to 80.2% and 73.1%, respectively, and 107× more energy efficient compared to Nvidia 1080 GTX. We demonstrate the silicon implementation of `PatterNet` and `TermiNETor` accelerators in TSMC40nm foundry enablement.

# Chapter 1

# Introduction

A modern integrated circuit (IC) consists of billions of densely packed transistors and convoluted metal tracks that connect these transistors. Electronic Design Automation (EDA) automates the tasks of design definition, planning, implementation, optimization, analysis, and validation. Integrated circuit design at the scale of billions of circuit elements would be unimaginable without the software and services from the EDA industry.

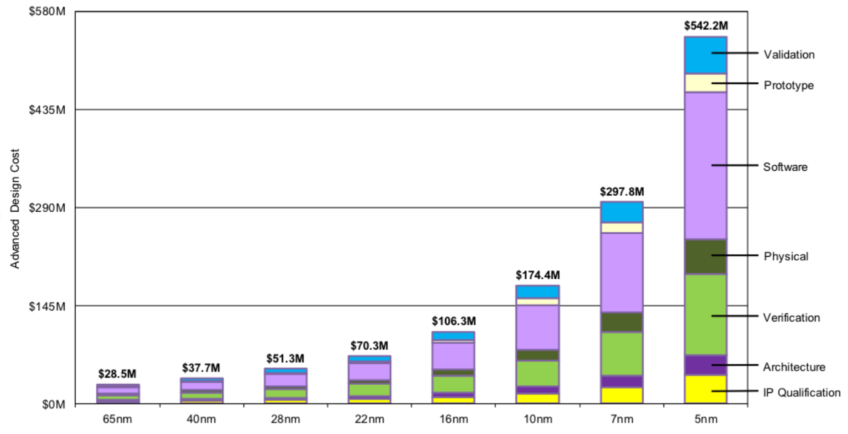


**Figure 1.1.** A major portion of the IC design cost is invested in the software tools that automate the design process [1]. Also, the software cost for a 5*nm* is nearly 100% larger than a 7*nm* chip.

However, the automation of integrated circuits comes at a cost. As shown in Figure 1.1, the majority of the advanced-node IC design cost is spent on the licenses of automation tools, making chip design very expensive. For example, the cost of designing a 3*nm* IC design ranges from $500 million to $1.5 billion [1]. The expensive tool licenses along with the long runtimes

[9] makes it prohibitively expensive for a rigorous design-space exploration.

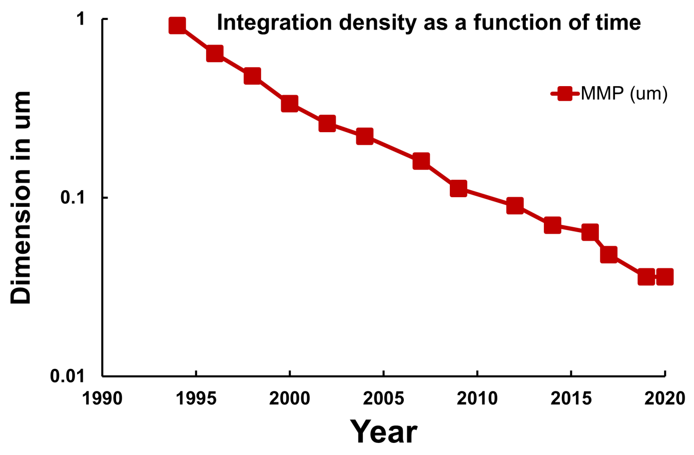


**Figure 1.2.** The reduction of minimum metal pitch (MMP) in recent years, indicates slowing of the regular doubling of integration density (Moore's law).

On the other hand, as miniaturization started to saturate [10] (Figure 1.2 shows the device scaling of Samsung foundry), the industry started to focus more on the innovations at the top of computing stack [11], such as assisting EDA algorithms with AI-based models and domain-specialized hardware architecture. In what follows, we discuss some of the promising opportunities within the algorithmic and hardware innovation areas. We explore and validate the extent of improvements available in these areas, using problems in the areas of IC design and neural-network acceleration. We argue that even if saturation of device scaling is stalling the performance gains, the innovations in algorithmic and specialized hardware architecture would continue to offer a viable way to deliver performance gains.

## 1.1 Machine Learning for Physical Design

Today, the two most important challenges in IC design are low PPA benefits when moving to newer nodes, and high design costs. While the stalling of Moore's Law manifests as reduced PPA gains at the bottom of the stack, the expensive per-launch tool license model limits the number of simultaneous tool launches, leading to limited design space exploration, which further impacts the design quality. With complex heuristics developed over three decades, the existing EDA algorithms (logic synthesis, placement, clock tree synthesis, and routing) have reached a point, where they are extremely mature and difficult to improve purely from the perspective of classical algorithmic techniques [12]. Machine learning has been proven to be commercially successful in areas of computer vision [13], robotics [14], finance [15], speech detection [16], video understanding [17] and anomaly detection [18]. We investigate the applications of machine learning for the challenging task of the physical design flow [19], as a powerful lever to extend the saturation of Moore's law. In particular, we focus on two key areas in which physical design subflows can benefit from the existing machine learning techniques: (a) estimators to predict the outcomes of timing analysis, and (b) active learning for improved placement quality.

### 1.1.1 Estimators for Timing Outcomes

The goal of learning-based estimators in EDA is to quickly predict tool outcomes and enable designers to drop unqualified designs. Dropping less-promising flow trajectories preserves compute resources and tool licenses, that can be spent elsewhere. For example, predicting downstream outcomes has proven to be useful in several phases [20] of the EDA flow; high-level synthesis [21], design verification [22, 23], wirelength [24], routing congestion [25], power, timing [26], voltage drop [27] [28], gate sizing [29, 30, 31] and lithography [32].

We develop learning-based techniques to predict STA tool outcomes. A significant portion of physical design runtime is spent on the analysis of timing across multiple process, voltage, and temperature (PVT) corners. At the same time, accurate, signoff-quality timing analysis is essential during place-and-route and optimization steps, to avoid loops in the flow as well as overdesign that wastes area and power.

### 1.1.2 Active Learning for Improved Placement Quality

Many problems in EDA are instances of design space explorations, where the goal is to search for an optimal (single- or multi-objective) design point in a design space. Reinforcement

Learning (RL) has proven to be a successful paradigm for exploratory learning, in performing a sequence of decisions to achieve goals, without the need for explicit training data.

Recent work has demonstrated the potential of learning-based methods for solving discrete optimization problems involving large solution spaces. Though some of the optimization problems in EDA can be modeled as a sequence of decisions, the underlying solution space is typically much larger than the traditional application domains of machine learning. For example, the solution space of the detailed placement problem involving 1000 standard cells and a placement canvas with 1000 sites is of the order $10^{2500}$, whereas Alpha-Go [33] has a state space of $10^{360}$. Prior work [34] has demonstrated the feasibility of applying RL to the placement problem, and the capability of deep networks to generalize representations across different designs. However, based on related works in RL for solving Combinatorial Optimization (CO) problems similar to placement [35, 36, 37], combining RL with existing combinatorial search algorithms has demonstrated promising results. These techniques leverage deep RL's ability to recover reasonable rough solution fast, and the ability of heuristics to realize greedy improvement via local refinement.

Markov Decision Process (MDP) describes a formal environment for RL. In MDP, there is an agent, capable of making choices of actions and these actions that lead to a state and reward outcome from the environment. Imagine having an explicit interaction between the agent and the environment that does not lead to a deterministic outcome of the state and reward. Meaning, the state transitions are stochastic, and the rewards are dependent on the state that the agent ends up in. Formally, MDP is a discrete-time state-transition system. It can be described with four components, (i) States (ii) Actions (iii) Transition model and (iv) Rewards. The accompanied examples refer to the standard cell placement task, which is central to the physical design flow.

- **States:** The states are necessary for choosing possible actions. For the chip placement task, the state could be the current locations of placed cells on a canvas.

- **Actions:** Next, we have a set of actions. These are chosen, in the simple case, from a small finite set. In the placement example, actions can be selecting the next cell to place, and choosing a location on the canvas, to place the next cell.

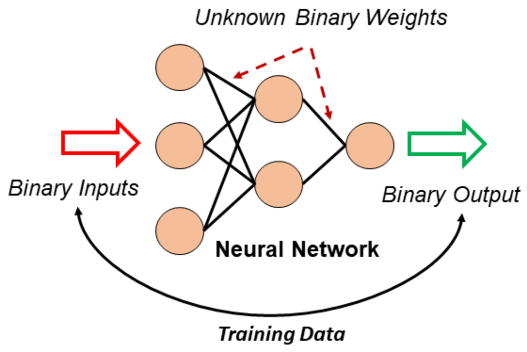
- **Markovian Transition model:** The transition probabilities describe the dynamics of the world. In scenarios, where it is difficult to represent the transition probability model explicitly, a simulator (or an EDA tool for the physical design sub flows) can be used to model the MDP implicitly by providing samples from the transition distributions.
- **Reward:** Reward is a short-term utility function that represents the goodness value of a state, from the agent's perspective. For the placement problem, reward could be a measure of wirelength and placement density.

**Policy:** A policy $\pi : S \to A$ is a global mapping or function from states to actions. In MDP, we are looking for one best policy among all possible solutions of policies.
**Value:** MDP can be expressed a huge tree. A value of a state keeps track of the expected discounted (with a discount factor $\gamma < 1$) long-term rewards of the state.

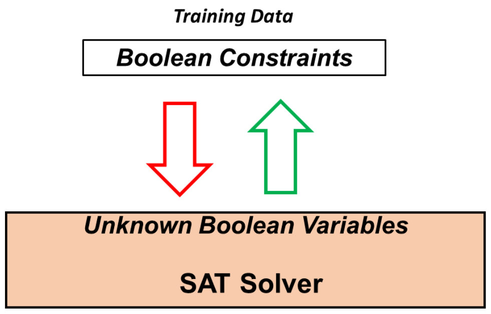
$$V^{\pi}(s) = \mathbb{E}_{\pi}\Big(\sum_{i=0}^{\infty} \gamma^{i} * R(s)\Big) \tag{1.1}$$

**Placement as a Markov Process:** The vanilla formulation of placement of rectangular cells on a rectangular, single-layer chip can be formalized as embedding the nodes of a graph - corresponding to a netlist - in the plane such that edge-lengths are minimized and nodes (cell instances) are subject to non-overlapping constraints. This can be concisely expressed as the following optimization problem over cell instance positions $(x_i, y_i)$ given the netlist-graph connectivity - a set of edges $\mathscr{E}$ - and edge-length parameters of the cells and chip - $l_i$ and $(b_x, b_y)$ respectively:

$$\begin{aligned} &\min_{x,y} \sum_{i,j \in \mathscr{E}} |x_i - x_j| + |y_i - y_j| \\ &\text{s.t. } \min\Big(\frac{l_i + l_j}{2} - |x_i - x_j|, \frac{l_i + l_j}{2} - |y_i - y_j|\Big) \leq 0 \\ &\quad |x_i| \leq (b_x - l_i)/2, \ |y_i| \leq (b_y - l_i)/2 \end{aligned} \tag{1.2}$$



**(a)** Binary neural networks.



**(b)** Boolean constraint satisfaction.

**Figure 1.3.** Relationship between the SAT problem and binary neural networks.

## 1.2 Boolean Satisfiability as a Function Fitting Problem

In the previous sections, we discussed machine learning algorithms for design analysis and optimization problems in IC design. Another important algorithmic paradigm is the domain of Boolean Satisfiability (SAT) solvers. The goal of Boolean Satisfiability, or the SAT problem, is to determine the 'satisfiability' of a Boolean set of equations for a set of inputs. The SAT problem can be extended to optimization problems, where the solvers find global optima without relying on gradients like the machine learning algorithms. Though the SAT solvers do not rely on gradients, these solvers can be imagined as a form of a complex function fitting exercise (similar to the machine learning tasks). Similar to binary neural networks, where weights and activations are Boolean values, the SAT solvers fit a model in which the model parameters are unknown variables and the training data represents a constraint on the values those parameters can possibly take.

In addition to the features of a SAT solver, the Satisfiability Modulo Theories (SMT) offer the option of optimizing one or more objective functions. In addition, the SMT frameworks support non-Boolean variables, various data structures, and predicate logic, thereby improving the richness of constraint and objective expression. SMT is well-suited to formulate joint optimization problems involving Boolean variables, integer variables, and real variables. In particular, complex conditional constraints can be succinctly and readily expressed in SMT. For example, an At-Most-One" (**AMO**) constraint over a set of Boolean variables imposes the requirement that at most one of the Boolean variables can be true (set to 1), and an "Exactly-One" (**EO**) constraint over a set of Boolean variables imposes the requirement that exactly one of the Boolean variables is true (set to 1). Another useful type of logical constraints is the predicate "If-Then-Else" (**ITE**$(a,b,c)$) constraint, which returns $b$ if $a$ is true and $c$ otherwise. Logical operations like conjunctions, disjunctions, or implications can be readily specified as well. Similar to the growth of machine learning libraries, the SAT competitions [38] alleviate the burden on the users of implementing their own search algorithm, thereby exploiting open-source SAT solvers that have shown tremendous progress over the last decade.

**SMT for Network-On-Chip Routing:** The routing problem is common at various levels of abstraction in the IC Design, from cell-routing inside the standard cells to network-routing of communication subsystems. In this thesis, we focus on the application-specific routing problem for network-on-chip (NOC). The routing canvas is represented as a grid, and the routing process is formulated using a multicommodity flow theory. Broadly speaking, the objective of any routing task is to find optimal source-destination routing paths on a routing grid, satisfying a set of constraints or rules. The built-in Boolean cardinality functions such as **EO** and **AMO** make it possible to succinctly capture flow conservation constraints and other logical implications. The built-in **ITE** function makes it possible to succinctly capture constraints like routing congestion on each routing link. These expressive modeling capabilities are essential in modeling our joint routing and VC assignment problem. Further, because modern SMT solvers like Z3 [39] are built on top of a Boolean satisfiability (SAT) solver, they are particularly good at navigating through search problems that are dominated by Boolean variables.

## 1.3 Domain Specialized Hardware

To fight the slowing of Moore's law, the computer architects have gradually shifted away from general purpose computing towards the new frontier of domain-specific hardware. Significant funding (> $ 5B) has been invested in the field of designing domain-specific AI hardware, to accelerate the performance and reduce the power consumption. Figure 1.4 shows the funding for 20 AI acceleration startups with more than $20M funding.

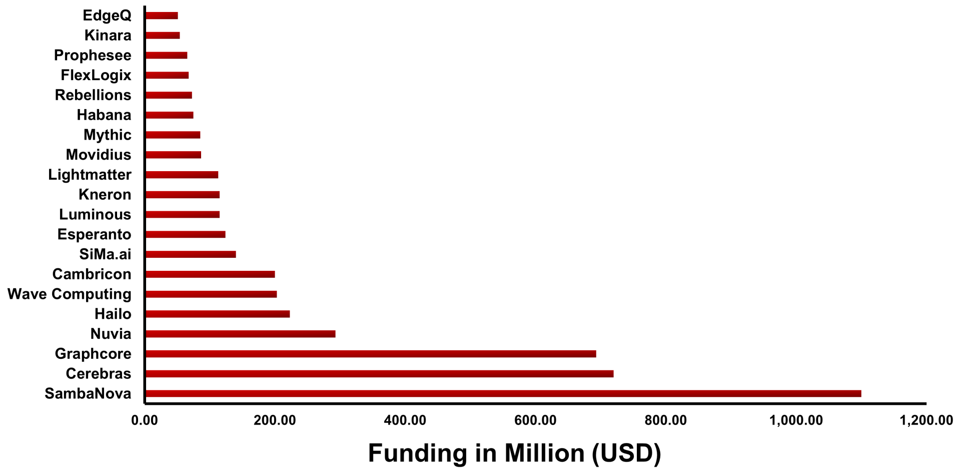


**Figure 1.4.** AI acceleration startups [2] with more than $50M funding as of November 2022.

### 1.3.1 Machine Learning Acceleration

Deep neural networks (DNN) are the state-of-the-art methods [40] for many real-world learning tasks, such as computer vision [13], natural language processing [41], and speech recognition [16]. The underlying learning algorithms are composed of two phases: training and inference. In the training phase, the training data is consumed by the network, and individual neurons assign a weighting to the input based on the task being performed (for example, edge detection in images). In the inference phase, the model uses the learned weights, to make predictions based on test data, eventually producing actionable results.

A key component of both the training and inference phases is matrix multiplication, that typically involves very large matrices. However, general purpose computing systems are far from optimal for this task of heavy-duty matrix multiplications. As compared to CPU and GPU, Google's ASIC TPU has far more multiplier units and more on-chip memory to store intermediate results. Figure 1.5 compares the performance of CPU, GPU and ASIC across six neural network applications [3]. The plot indicates rooflines for CPU and GPU are consistently below their ceilings than was the TPU. In this dissertation, we focus on accelerating the inference phase of neural network based image classification tasks. To alleviate the computational and energy burden of neural network inference, we propose two accelerators `TermiNETor` and `PatterNet` to exploit (i) reuse of computations, and (ii) skipping of ineffectual computations. To fully exploit the available compute resources, an overarching goal of the ASIC-based accelerators [42, 43] is to keep the compute units in constant use. Therefore, we propose data flows and the supporting hardware for efficient data reuse of weights and activations, and thereby exploiting the full potential of our AI accelerators. We also demonstrate the silicon implementation of our accelerators in TSMC 40nm foundry enablement.

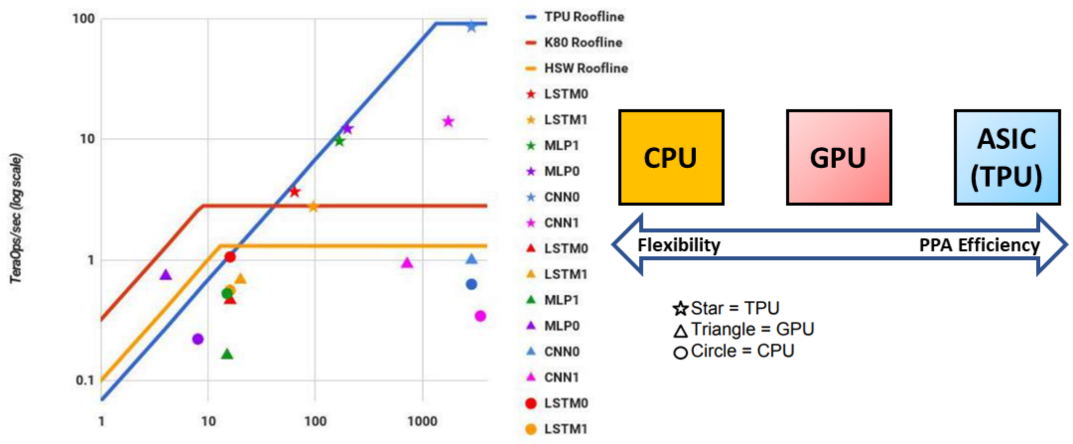


**Figure 1.5.** The Roofline model for TPU ASIC (blue) has much longer slanted line as compared to NVIDIA K80 GPU (red) and Intel Haswell CPU (yellow) [3].

## 1.4 Thesis Outline

To fight the stalling of performance gains from device scaling, the dissertation explores innovations at the top of the computing stack. To quantify the performance gains, we use some of the promising problems in physical design, communication-subsystem networks, and domain-specialized hardware. The interactions between AI, EDA and ASIC Design worlds is neatly summarized in Figure 1.6. To elaborate, AI has proven its success in multiple applications, of which EDA is an important specialty. Similarly, ASIC Design serves multiple applications, of which AI is a key area of interest today. The trivial usecase of EDA flows for ASIC Design forms a cyclic dependency between the three worlds.

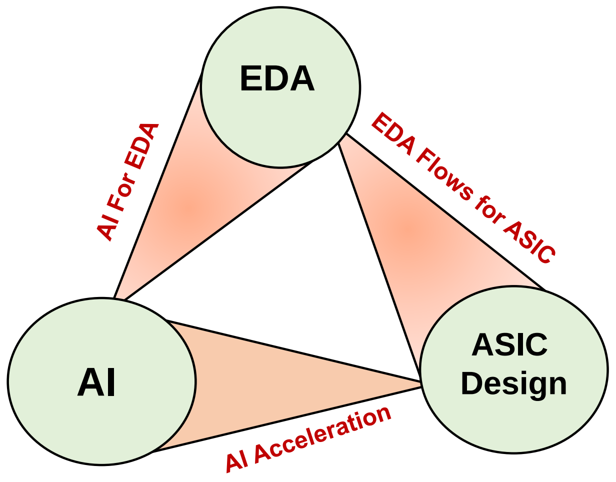


**Figure 1.6.** Dependencies between AI, EDA and ASIC Design. Inspired from Roger Penrose's metaphysical framework [4].

The contributions of this thesis are summarized as follows:

- In Chapter 2, we explore the applications of machine learning to predict timing outcomes in the physical design flow. We propose a learning model to predict expensive PBA results from GBA results, essentially predicting PBA results for "free". Also, we present another learning model, to accurately capture and exploit the physics of multi-corner timing correlation, to improve design convergence, and reduce design cost.

- In Chapter 3, we formulate the placement problem as a Markov Decision Process at a coarse-level (cell-clusters), to find optimal sequences of cell-cluster movements. The idea is to leverage deep RL's ability to determine the cluster positions reasonably fast, and exploit the ability of SAT solvers for further local refinement.

- In Chapter 4, we use SMT to jointly solve the routing and static virtual channel allocation in network-based communication systems, to enable better deadlock-free performance of the network. Considering both the routing and VC allocation simultaneously allows for greater flexibility in channel selection compared to prior state-of-the-art application-aware oblivious routing algorithms, which ultimately leads to better performance.

- In Chapter 5, we propose a novel neural network accelerator `TermiNETor` to dynamically predict and terminate ineffectual computations during the inference phase of image classification tasks, thereby alleviating the computation and energy burden. The early termination of ineffectual computations in `TermiNETor` framework leads to better energy efficiency as compared to the GPU and other state-of-the-art sparsity-aware accelerators.

- In Chapter 6, we present another neural network accelerator `PatterNet` that is based on optimizing computation reuse during the inference phase of image classification tasks. The sharing of cluster topologies between filters in the proposed `PatterNet` framework results in better reuse of computations and reduced memory footprint, eventually translating in to area, performance and energy benefits as compared to the state-of-the-art accelerators.

- Chapter 7 summarizes the contributions of this dissertation, and presents possible directions for future work.

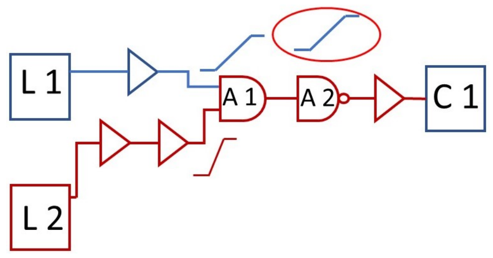
# Chapter 2

# AI for EDA: Design Analysis

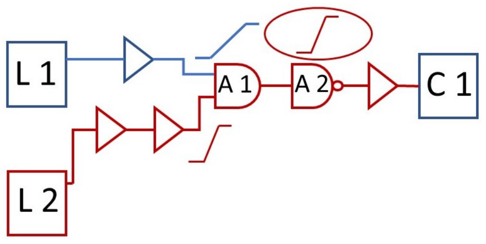
## 2.1 Introduction

Long runtimes of modern Electronic Design Automation (EDA) tools for designs with over a million instances and many multi-corner multi-mode (MCMM) timing scenarios block quick turnaround time in system-on-chip (SOC) design. A significant portion of runtime is spent on analysis of timing across multiple process, voltage and temperature (PVT) corners. At the same time, accurate, signoff-quality timing analysis is essential during place-and-route and optimization steps, to avoid loops in the flow as well as overdesign that wastes area and power.

**Graph-based and Path-based Tradeoff:** To support PBA-GBA tradeoff, STA tools such as [44] and [45] support graph-based (GBA) and path-based (PBA) modes in static timing analysis (STA), enabling a tradeoff of accuracy versus turnaround time. In GBA mode, pessimistic transition time is propagated at each node of the timing graph. Figure 2.1(a) illustrates transition propagation from launch flip-flop L2 to capture flip-flop C1. At instance A1, the worst of its input transition times is propagated from input to output. However, the worst transition time happens to be on the pin that is not part of the L2-C1 timing path. Since cell delay estimation is a function of input transition time, the GBA-mode delay calculation for instance A2 is performed using a pessimistic transition time. This pessimism accumulates along the timing path, leading to pessimistic arrival time calculation at the endpoint. Further, the transition time at the endpoint influences the setup requirement of flip-flop C1, adding further pessimism to the reported slack of the timing path.



**(a)**



**(b)**

**Figure 2.1.** Transition propagation in (a) GBA mode and (b) PBA mode.

In PBA mode, path-specific transition time is propagated at each node of the timing graph. Figure 2.1(b) illustrates transition propagation for the L2-C1 timing path in PBA mode. For instance A1, actual path-specific transition time is propagated, and is therefore used in the cell delay calculation for A2. As the number of timing paths to an endpoint increases, there is an exponential increase in possibilities of transition propagation and delay calculation at each node.[1] The path-specific transition propagation and arrival time estimation at each node is runtime-intensive. Figure 2.2 shows that for public benchmark designs [46] [47] [48] implemented in a 28nm FDSOI foundry enablement, a commercial signoff STA tool exhibits slowdowns (PBA runtime, relative to GBA runtime) as high as 15X for leon3mp [48] (108K flip-flops, 450K signal nets), and 150X for megaboom [47] (350K flip-flops, 960K signal nets).[2] The need for faster path-based analysis is called out in, e.g., Molina [9]; Kahng [49] names prediction of PBA slacks from GBA analyses as a key near-term challenge for machine learning in IC implementation.

Modern IC implementation in advanced nodes relies on the expensive PBA mode for signoff. Thus, if PBA were to be "free" (i.e., without any runtime or other overheads) relative

[1]Details of PBA are given in proprietary tool documentation of major EDA vendors, and analysis outcomes typically have subtle differences across tools.

[2]This slowdown is seen with "exhaustive" and "slack_greater_than -1" PBA, which assures accuracy in the path-based analysis and provides the least pessimistic basis for optimization. Because runtimes are so long with exhaustive mode, users must typically use "path" mode in which path-specific timing recalculation is performed only for some set of timing paths. Analysis using path mode does not guarantee to report worst possible paths to a given endpoint of interest, but can have as little as 2X runtime overhead (pba_mode path and nworst 1) versus GBA.

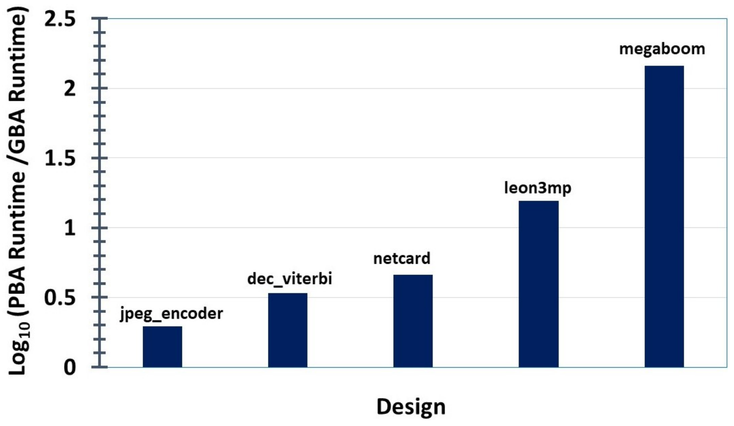


**Figure 2.2.** Ratio of PBA runtime to GBA runtime on log scale (commercial signoff STA tool; 28nm FDSOI foundry enablement) for public-domain design examples (see Table 2.5 for details) ranging in size from 530 flip-flops and 11K signal nets to 350K flip-flops and 960K signal nets.

to GBA, then only PBA would be used. Unfortunately, today's PBA runtime overheads force the design methodology to make difficult accuracy-runtime tradeoff choices. If there is a high timing violation count in early phases of physical implementation, timing analysis accuracy may not be a primary concern. Hence, designers will typically use less-accurate but relatively inexpensive GBA mode in the early stages of design. Later in the design cycle, as the design converges towards fewer violations, designers must enable PBA mode, at a minimum for timing paths which fail in GBA mode, so as to obtain less-pessimistic, path-specific timing slacks and prevent over-fixing.[3] However, by this time, damage has already been done to the design's power and area metrics, as a result of performing GBA-driven optimizations.

Figure 2.3 illustrates the magnitude of PBA-GBA divergence using a commercial signoff timer, for the megaboom testcase implemented in 28nm FDSOI. One worst GBA path is extracted per endpoint (corresponding to "nworst 1" in commonly-used STA tool Tcl), and the top 15K timing paths are plotted in decreasing order of PBA-GBA divergence (by the nature of PBA and GBA, the latter is always pessimistic with respect to the former). The maximum PBA-GBA divergence of 110ps means that the GBA can be pessimistic by 110ps as compared to PBA, for this testcase. Predicting PBA "for free" can enable improvement of design quality and schedule,

[3]The turnaround time overheads of PBA are compounded by having many MCMM scenarios in timing closure during final stages of implementation, especially for low-power, high-performance designs in advanced nodes.

independent of timing slack value. This strongly motivates our present work to develop a fast predictor of PBA from GBA analysis.

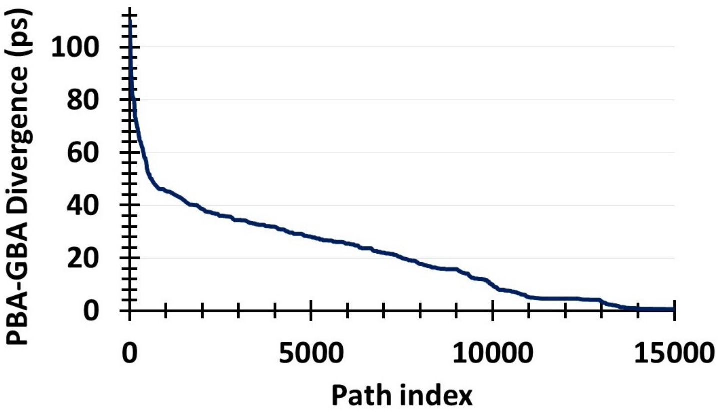


**Figure 2.3.** PBA-GBA divergence for the megaboom design (350K flip-flops, 960K signal nets) signed off at 1.2ns clock period in 28nm FDSOI technology.

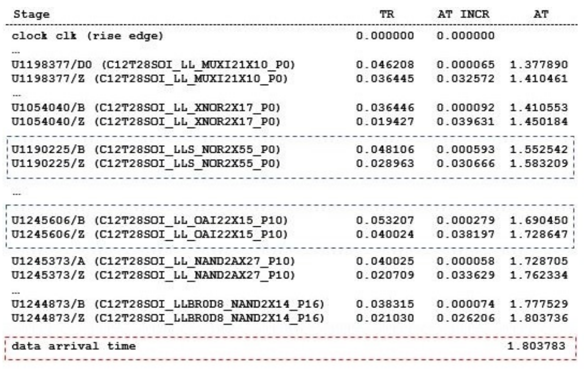
**Unobserved Corner Prediction:** A substantial amount of tool runtime, as well as overall compute and license resources, must be spent to obtain analysis of timing across multiple process, voltage and temperature (PVT) analysis *corners*. All else being equal, design teams would like to always have accurate analyses at all signoff corners: this helps avoid expensive loops in the place-and-route and optimization steps of the flow, and also helps avoid overdesign that wastes area and power. However, with well over 100 signoff analysis corners in advanced-node IC design, as a practical matter designers can analyze timing at only a small number of corners during most of the design steps leading up to final signoff. This potentially masks many real violations, since PVT effects on signal arrival time at a given timing endpoint will depend on the *combination* of gates and wires in any given timing path. To reduce the risk of masking real violations that are realized only in final timing signoff runs, designers add flat timing margins or increase the target clock period. However, there is no canonical methodology of determining the best mix of analyzed corners, padding by flat margins, relaxation of frequency targets, etc. – all of which eventually result in overdesign.

If golden timing analysis across all signoff corners were to be "super-fast", with minimal schedule and compute resource overheads, designers would be able to reduce overdesign, and detect and fix real timing violations much earlier in the design cycle. Our present work pursues a data-driven approach to this challenge of reducing timing analysis effort in advanced-node IC design, with the aim of enabling faster design convergence. A motivating observation is that path delays at different corners are strongly correlated, with correlations dependent on the topology and structure of the timing path. Our work seeks to accurately capture and exploit this correlation, to improve design convergence and reduce schedule cost.

## 2.2 Related Work

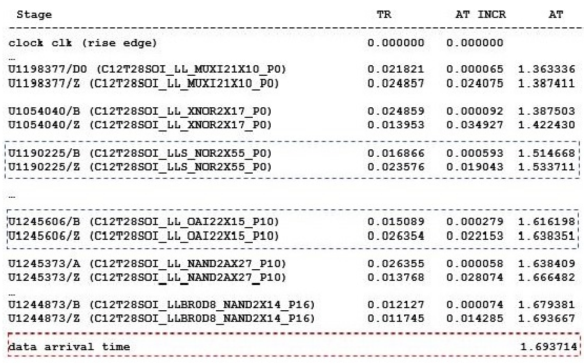
Recent works such as [20, 49] highlight the potential use of machine learning to achieve faster design convergence. Speed, accuracy and scalability of STA has for decades been a focus of industry R&D attention; see, e.g., TAU Workshop [50], [51] and [52]. Prior works on the use of machine learning to reduce miscorrelation between STA engines or long optimization runtimes are given by [53], [54], [55], and [56] seeks to reduce STA runtime itself through distributed computing. Kahng et al. [57] provide methodology to model signal integrity (SI) effects on path arrival time, using machine learning. Since their model [57] predicts SI from non-SI, the degree of pessimism in SI prediction is not a primary concern. However, as noted above, model prediction is a serious concern when predicting PBA from GBA, since an optimistic PBA prediction might mask a real timing violation. A quick predictor of PBA from GBA can potentially reduce the overdesign and the design schedule cost.

On the other hand, learning-based STA prediction has been previously studied by [58] and [59], e.g., the latter uses linear regression, SVM and random forest models to account for dynamic NBTI aging and other correlated on-chip variations. Methods that determine a smaller subset of analysis corners, to improve STA runtimes, have been actively pursued. An important work of Onaissi et al. [60] focuses on hold time analysis and determining a minimum set of *dominant corners* (whose satisfaction will result in timing feasibility at all other corners); the authors apply an additional *dominance margin* to reduce the size of this dominance set. Silva et



```
Stage                                                TR        AT INCR     AT
---------------------------------------------------------------------------------
clock clk (rise edge)                                0.000000  0.000000
...
U1198377/D0 (C12T28SOI_LL_MUXI21X10_P0)              0.046208  0.000065  1.377890
U1198377/Z (C12T28SOI_LL_MUXI21X10_P0)               0.036445  0.032572  1.410461
...
U1054040/B (C12T28SOI_LL_XNOR2X17_P0)                0.036446  0.000092  1.410553
U1054040/Z (C12T28SOI_LL_XNOR2X17_P0)                0.019427  0.039631  1.450184

U1190225/B (C12T28SOI_LLS_NOR2X55_P0)                0.048106  0.000593  1.552542
U1190225/Z (C12T28SOI_LLS_NOR2X55_P0)                0.028963  0.030666  1.583209

...

U1245606/B (C12T28SOI_LL_OAI22X15_P10)               0.053207  0.000279  1.690450
U1245606/Z (C12T28SOI_LL_OAI22X15_P10)               0.040024  0.038197  1.728647

U1245373/A (C12T28SOI_LL_NAND2AX27_P10)              0.040025  0.000058  1.728705
U1245373/Z (C12T28SOI_LL_NAND2AX27_P10)              0.020709  0.033629  1.762334
...
U1244873/B (C12T28SOI_LLBR0D8_NAND2X14_P16)          0.038315  0.000074  1.777529
U1244873/Z (C12T28SOI_LLBR0D8_NAND2X14_P16)          0.021030  0.026206  1.803736

data arrival time                                                        1.803783
```

**(a)**



```
Stage                                                TR        AT INCR     AT
---------------------------------------------------------------------------------
clock clk (rise edge)                                0.000000  0.000000
...
U1198377/D0 (C12T28SOI_LL_MUXI21X10_P0)              0.021821  0.000065  1.363336
U1198377/Z (C12T28SOI_LL_MUXI21X10_P0)               0.024857  0.024075  1.387411

U1054040/B (C12T28SOI_LL_XNOR2X17_P0)                0.024859  0.000092  1.387503
U1054040/Z (C12T28SOI_LL_XNOR2X17_P0)                0.013953  0.034927  1.422430

U1190225/B (C12T28SOI_LLS_NOR2X55_P0)                0.016866  0.000593  1.514668
U1190225/Z (C12T28SOI_LLS_NOR2X55_P0)                0.023576  0.019043  1.533711

...

U1245606/B (C12T28SOI_LL_OAI22X15_P10)               0.015089  0.000279  1.616198
U1245606/Z (C12T28SOI_LL_OAI22X15_P10)               0.026354  0.022153  1.638351

U1245373/A (C12T28SOI_LL_NAND2AX27_P10)              0.026355  0.000058  1.638409
U1245373/Z (C12T28SOI_LL_NAND2AX27_P10)              0.013768  0.028074  1.666482
...
U1244873/B (C12T28SOI_LLBR0D8_NAND2X14_P16)          0.012127  0.000074  1.679381
U1244873/Z (C12T28SOI_LLBR0D8_NAND2X14_P16)          0.011745  0.014285  1.693667

data arrival time                                                        1.693714
```

**(b)**

**Figure 2.4.** PBA-GBA divergence for a design signed off at 1.2ns ( 990K standard cells) in 28nm FDSOI technology. Shown: timing analysis for the same path in (a) GBA mode and (b) PBA mode.

al. [61] give a branch-and-bound methodology for identifying exactly a *single* corner that has worst delay. To avoid the analysis corner explosion entirely, Onaissi and Najm [62] propose what is effectively a "cornerless" approach that uses a single run of STA, with propagation of delay and slew models that are linear functions of process parameters, to cover all process corners. A large literature (e.g., [63]) has investigated statistical STA, which can effectively mitigate both corner explosion and the increasingly dominant impacts of manufacturing variability and low-voltage operating modes. We seek to evaluate the feasibility of learning-based prediction of timing results at many *unknown* corners, based on timing results from comparatively few *known* corners.

## 2.3 PBA-GBA: Preliminaries

Table 2.1 lists terms and definitions used for the PBA-GBA modeling framework. Formally, our problem is: Given a training set $N$ of (PBA-GBA) path-consistent path analysis pairs, such as the pair shown in Figure 2.4, use $N$ to train a learning model that predicts PBA-GBA divergence for a set $T$ (where $T \cap N = \emptyset$) of paths that are analyzed in only GBA mode.

**Table 2.1.** Terms and definitions for PBA-GBA.

| **Term** | **Definition** |
|---|---|
| *Bigram* or *bigram unit* | Two consecutive (cell) stages in a timing path |
| $AT$ | Arrival time |
| $TR$ | Transition time |
| $PD$ | Propagation delay |
| $SL$ | Timing slack |
| $C_L[j]$ | Load capacitance of a driving instance (cell) $j$ |
| $F_O[j]$ | Fanout of driver cell $j$ |
| $DR[j]$ | Drive strength of instance $j$ |
| $N_m$ | Number of stages in a timing path |
| $N_s[j]$ | Stage depth of instance $j$ relative to launch flop |
| $N_{bg}$ | Number of bigrams in a timing path |
| $G[j]$ | (Logical) functionality of an instance $j$ |
| $AT_{gba}[i,j]$ ($AT_{pba}[i,j]$) | AT of instance $j$, pin $i$ in GBA (PBA) mode |
| $TR_{gba}[i,j]$ ($TR_{pba}[i,j]$) | TR of instance $j$, pin $i$ in GBA (PBA) mode |
| $TR\_MAX_{gba}[j]$ | $\max_i\{TR_{gba}[i,j]\}$ |
| $\Delta TR[i,j]$ | $TR_{gba}[i,j] - TR_{pba}[i,j]$ |
| $TR\_rat_{gba}[i,j]$ | $1 - \frac{TR_{gba}[i,j]}{TR\_MAX_{gba}[i,j]}$ (TR ratio) |
| $Acc\_TR\_rat_{gba}[i,j]$ | $\sum_0^{N_s[j]} TR\_rat_{gba}[i,j]$ (accum. TR ratio) |
| $PD_{gba}[j]$ ($PD_{pba}[j]$) | PD of an instance $j$ in GBA (PBA) mode |
| $SL_{gba}[j]$ ($SL_{pba}[j]$) | SL of an endpoint $j$ in GBA (PBA) mode |
| $\Delta PD[j]$ | $PD_{gba}[j] - PD_{pba}[j]$ |
| $\Delta AT[i,j]$ | $AT_{gba}[i,j] - AT_{pba}[i,j]$ |
| $\Delta SL[j]$ | $SL_{gba}[j] - SL_{pba}[j]$ |

## 2.3.1 Intuition for Bigram-Based Modeling

PBA-GBA divergence of a timing path can be estimated either *stage-wise* or *path-wise*. We refer to the latter approach as *lumped* path modeling. For stage-wise modeling, $n \geq 1$ consecutive stages in a timing path are termed an *n-gram* or *n-gram unit* within the path. As $n$ increases, stage-wise modeling (by $n$-gram units) approaches lumped modeling. The definitions of PBA-GBA divergence in Section 5.1 straightforwardly extend to stage-wise modeling. The accumulation of PBA-GBA divergence over the $n$-grams in a path leads to a path-specific PBA-GBA divergence. Figure 2.5 shows a bigram-based ($n = 2$) representation of a timing path from launch flip-flop L1 to capture flip-flop C1.

Based on numerous preliminary studies, we have chosen stage bigrams as the fundamental unit for our modeling approach. We observe that lumped modeling shields stage-specific details and inter-stage variations, and is prone to large optimistic errors (in our attempts). The lumped approach has a very large space of parameters (which can grow with the number of stages) available to characterize a given path. In addition, it is difficult to locate outlier stages that

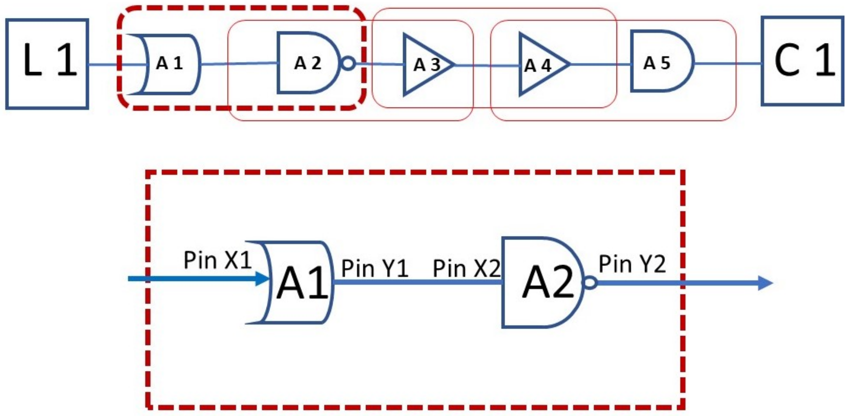


**Figure 2.5.** Bigram-based model of a timing path. The timing path is represented as a series of four bigram units.

are the “root causes” of misprediction. By contrast, stage-based modeling ensures a fine-grain modeling for each stage and accounts for inter-stage variation of circuit parameters. Since path prediction is an accumulation of stage predictions, bounding stage-wise errors helps limit path mispredictions. (Practically, it is also easier to identify and diagnose outliers in a stage-based model, and to improve the model by added features that reduce mispredictions for these outliers.)

We also observe that PBA-GBA divergence arises from the existence of ‘competing’ transition values at inputs of a cell. However, this will be translated into arrival time divergence only for the *next* stage in a given timing path. This naturally motivates use of a bigram as the basic modeling unit to capture PBA-GBA divergence. We find that the training of *n*-gram models for $n > 2$ is hampered by the combinatorial explosion of possible *n*-grams (e.g., over a given cell library), while training of bigram models to achieve accurate prediction is computationally more tractable.

## 2.3.2 Selection of Features

We have evaluated a comprehensive set of electrical and physical parameters of a bigram unit that can affect PBA-GBA divergence. Our analyses indicate that transition time in GBA mode $TR_{gba}$ at the primary input of a bigram unit, and transition time ratio in GBA mode $TR\_rat_{gba}$, are two mandatory parameters that strongly impact PBA-GBA divergence of the bigram unit. However, these two parameters alone are insufficient for accurate prediction of PBA-GBA divergence. Parameters such as cell drive strength and gate type influence the transition time variation at the input cell which is reflected at the bigram output pin. In addition, arrival time of the bigram unit is an indicator of the positioning of the bigram unit along the timing path: the arrival time reflects topological distance from the launch flip-flop, as well as parametric on-chip variation (POCV) derating and error propagation along the timing path. Other layout-dependent electrical parameters such as output load capacitance and fanouts of the cells in the bigram unit are also found to be useful in predicting PBA-GBA divergence. In total, the bigram-based model for which we report results below uses the following 13 parameters extracted from GBA analysis:

1. transition time of the first cell in the bigram unit;
2. transition time of the second cell in the bigram unit;
3. arrival time of first cell in bigram unit;
4. transition time ratio (TR) of first cell in the bigram unit;
5. arrival time of second cell in bigram unit;
6. drive strength of first cell in bigram unit;
7. drive strength of second cell in bigram unit;
8. functionality of first cell in bigram unit;
9. functionality of second cell in bigram unit;
10. fanout of first cell in bigram unit;
11. load capacitance of first cell in bigram unit;
12. accumulated transition time ratio of first cell in bigram unit;
13. propagation delay of second cell in bigram unit.

Our studies indicate that dropping any one of these parameters reduces absolute model accuracy by at least 2%, and dropping any two parameters at a time reduces the model accuracy by at least 4%. Here, we define the *mean_initial* accuracy as the mean of absolute (Predicted − Actual) arrival times, with all 13 parameters (features) used. The *mean_reduced* accuracy is the mean of absolute (Predicted_New − Actual) arrival times, with reduced features. Then, we define the *model accuracy reduction* (%) as (mean_reduced − mean_initial) × 100 / (mean_initial). Figures 2.6(a) and (b) illustrate the sensitivity of accuracy reduction to particular parameters. Accuracy with none of the features dropped is used as baseline for comparison. Dropping $TR\_rat_{gba}$ alone reduces the model accuracy by 27%, and dropping any pair combination that includes $TR\_rat_{gba}$ corresponds to the largest accuracy reductions in Figure 2.6(b).

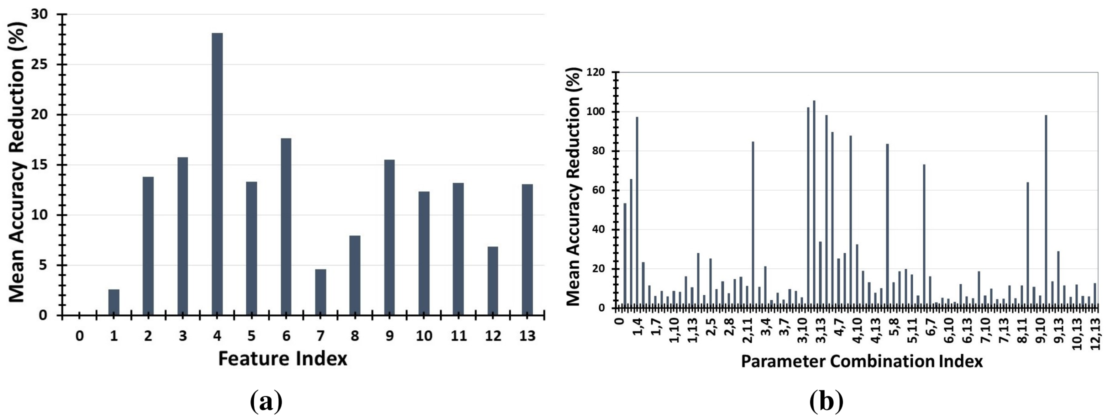


**Figure 2.6.** The impact of (a) Dropping any single one of the 13 parameters (indexed as above); the peak loss of accuracy corresponds to $TR\_ret_{gba}$. (b) Dropping any pair of the 13 parameters at a time; the x-axis gives, from left to right, C(13,2) = 78 pairs (1,2), (1,3), ..., (12,13).

Last, since PBA-GBA divergence depends on the incremental transition time for each bigram unit, we find that it is necessary to implement a *two-phase* modeling strategy: (i) Phase 1 predicts incremental transition time gain in PBA mode, and (ii) Phase 2 uses predictions from Phase 1 along with other features from GBA mode analysis to predict PBA-GBA divergence for the bigram unit. We give more details of this two-phase strategy in the next section.

### 2.3.3 Classification and regression trees

*Classification and regression trees* (CART) [64] are nonlinear techniques for constructing predictive models from data. The models are obtained by recursively partitioning the data space into feature space and fitting a simple predictive model within each partition. This recursive partitioning can model a data set with complex feature interactions. In the context of PBA-GBA prediction, *regression trees* can be used to predict PBA arrival time for each bigram unit, i.e., we use features of the test data (GBA analysis results) to predict PBA arrival time for each data point. *Classification trees* can predict PBA-GBA divergence (incremental arrival time gain) where the model uses features of the test data to predict PBA-GBA divergence for each data point.

An important realization is that since regression tree-based modeling is limited by the span of PBA arrival time values in the training data, testing is always constrained by the range of arrival time values covered in training phase. On the other hand, classification tree-based modeling is limited by the range of PBA-GBA arrival time increments used in training data. During testing, if a data point exceeds the class value used in training data, the model is constrained by the span of increments in the training data.

As an example, consider a training data set with GBA and PBA arrival time ranges of 24ps to 345ps, and 14ps to 326ps, respectively, along with PBA-GBA divergence range of 0ps to 40ps. In regression-based modeling, a test data point with GBA arrival time of 560ps is constrained by the span of training data, which is 345ps in this case. In classification-based modeling, predicted PBA-GBA divergence will be from one of the values in the range of 0ps to 40ps. If we ensure that the span of possible PBA-GBA divergence values are covered in the training data, mispredictions can be reduced. In addition, having positive class values gives us "sensibility by construction" in our predictions, since actual PBA-GBA divergence can never be negative. Our preliminary studies, summarized in Table 2.2 for the netcard testcase, lead us to use the classification tree approach for PBA-GBA divergence prediction.

**Table 2.2.** Regression versus classification trees (netcard).

| Metric | Regression | Classification |
|---|---|---|
| $model\_opt_{path}$ | 18.65ps | 8.21ps |
| $model\_99p_{path}$ | 12.96ps | 6.44ps |

## 2.4 PBA-GBA: Modeling Methodology

After selection of parameters, our modeling methodology includes application of machine learning techniques that capture complex interactions of the parameters and their impact on PBA-GBA divergence. We find that linear regression techniques fail to capture nonlinearity of predictions and complex interactions between parameters. For example, interaction of parameters such as input transition, output load and cell drive strength influence PBA-GBA divergence. We have also evaluated nonlinear modeling techniques such as multivariate adaptive regression splines (MARS) [65] which suffer from two-sided distribution of error. Since PBA is always optimistic as compared to GBA, a pessimistic prediction (i.e., prediction of less timing slack than the given GBA slack) is incorrect. With bigram-based modeling, as the number of data points used for modeling increases (1M+), our results indicate that MARS is not scalable when higher-order effects are introduced. Ultimately, for improved accuracy, reduced variance and faster runtimes, we have focused our efforts on tree ensemble methods. Random forests of classification trees give the best results so far, and Figure 2.7 illustrates the visual aid inherent in tree-based modeling, which helps to better understand feature importance and classification criteria. We discuss more about classification and regression trees in Section 2.3.3.

### 2.4.1 Reporting Metrics

PBA-GBA divergence signifies the pessimism in GBA mode. Therefore, reduction in this pessimism is an appropriate metric to evaluate the predictive model. Figure 2.8 shows a path-consistent plot of actual GBA versus PBA path arrival times. The maximum PBA-GBA

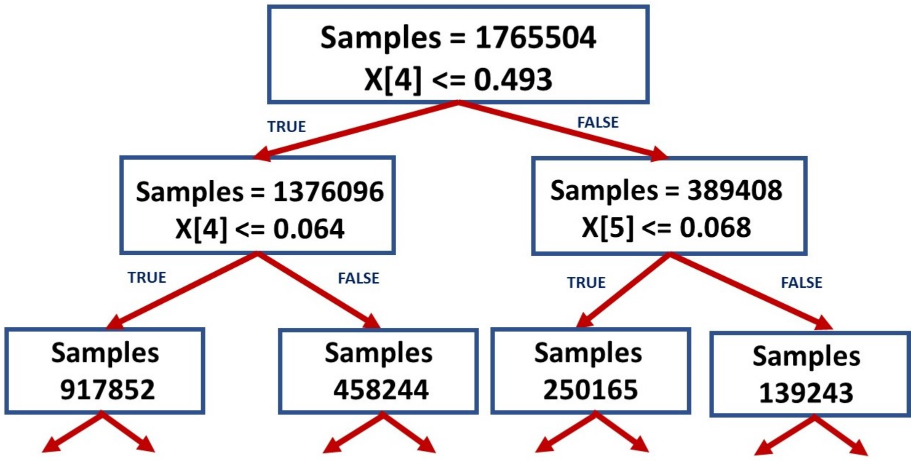


**Figure 2.7.** Tree-based classification with 1.7M training samples and 13 parameters. Parameter X[4], which corresponds to $TR\_rat_{gba}$, splits the data space into 75% and 25% with a split value of 0.493, indicating its importance in classifying input data.

divergence is 110ps. The blue band signifies the pessimism in GBA mode as compared to PBA mode. The intent of machine learning-based PBA prediction is to reduce width of the blue band in a predicted PBA versus actual PBA plot. Ideally, the plot of predicted PBA versus actual PBA would be the straight orange line $Y = X$, i.e., zero pessimism in the prediction.

Table 2.3 shows actual PBA-GBA divergence metrics from a commercial signoff timer that we use as the reference to quantify the accuracy of our predictive model.

**Table 2.3.** PBA-GBA divergence metric.

| Notation | Meaning |
|---|---|
| $actual\_max_{path}$ | Upper bound of actual PBA-GBA divergence |
| $actual\_99p_{path}$ | 99th percentile value of sorted PBA-GBA divergence (in ascending order) |
| $actual\_mean_{path}$ | Mean absolute value of actual PBA-GBA divergence |

Table 2.12 explains path-consistent and endpoint-consistent divergence metrics that we define for model predictions. The extent of reduction of these metrics, as compared to reference PBA-GBA divergence metrics, signifies the reduction of pessimism as compared to GBA of a commercial timer.

**Table 2.4.** Model prediction-based divergence metrics.

| Notation | Meaning |
|---|---|
| **Path-consistent prediction metrics** | |
| $model\_max_{path}$ | Worst-case pessimistic prediction divergence |
| $model\_opt_{path}$ | Worst-case optimistic prediction divergence |
| $model\_99p_{path}$ | 99th percentile value of absolute prediction divergence values (in ascending order) |
| $model\_mean_{path}$ | Mean absolute value of prediction divergence values |
| **Endpoint-consistent prediction metrics** | |
| $model\_max_{end}$ | Worst-case pessimistic prediction divergence |
| $model\_opt_{end}$ | Worst-case optimistic prediction divergence |
| $model\_99p_{end}$ | 99th percentile value of absolute prediction divergence values (in ascending order) |
| $model\_mean_{end}$ | Mean absolute value of prediction divergence values |

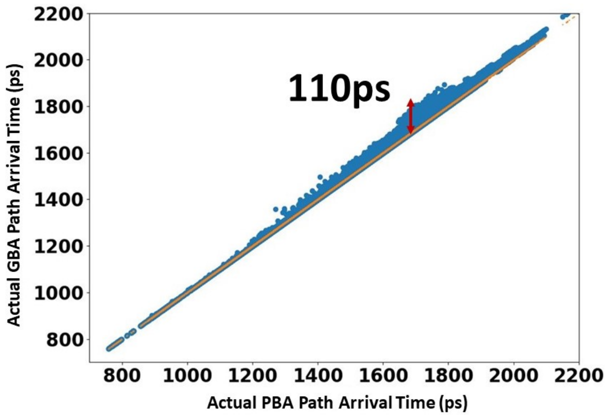


**Figure 2.8.** PBA versus GBA path-consistent arrival time plots using a commercial timer in 28nm FDSOI technology.

### 2.4.2 Model Definition

For Phase 1 and Phase 2 of our modeling, equations 2.1 and 2.2 capture PBA-GBA divergence in transition time and arrival time respectively, for each bigram unit.

$$\Delta TR_{bg} = f(C_L, DR, G, F_O, TR_{gba}, AT_{gba}, TR\_rat_{gba}, Acc\_TR\_rat_{gba}) \qquad (2.1)$$

$$\Delta AT_{bg} = f(C_L, DR, G, F_O, TR_{gba}, AT_{gba}, TR\_rat_{gba}, Acc\_TR\_rat_{gba}, \Delta TR_{bg}) \tag{2.2}$$

PBA-GBA divergence for a timing path is estimated by cumulative addition of bigram PBA-GBA divergence values in the timing path. This is explained in Equation 3.2

$$\Delta AT_{path} = \sum_{1}^{N_{bg}} \Delta AT_{bg} \tag{2.3}$$

### 2.4.3 Modeling Flow

We propose a modeling flow as illustrated in Figure 5.8. During the model training, both GBA and PBA path-consistent timing reports are used as inputs. We then extract parameters required to model PBA-GBA divergence for each bigram pair. During the model testing, the model predicts PBA-GBA divergence for any unseen (i.e., new) GBA timing path. Predicted PBA timing results that are output by our model can subsequently serve as, e.g., inputs to optimization and sizing steps of the physical implementation flow.

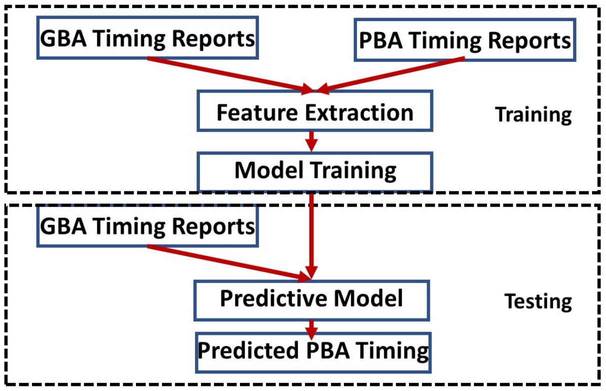


**Figure 2.9.** Our modeling flow.

## 2.5 PBA-GBA: Experimental Validation

We now describe our design of experiments to validate the predictive model. For each of these experiments, we discuss our modeling results.

### 2.5.1 Design of Experiments

In our experiments, we use in-house developed artificial designs and five real designs as listed in Table 2.5. We use 28nm FDSOI foundry technology libraries for all our experiments. The training time for our model on an Intel Xeon 2.6 GHz server is 219 seconds, for a training data set with 2.44M bigrams and time for testing phase is 17 seconds for test data with 1.04M bigrams.

**Table 2.5.** Design data used for experiments.

| Design | # Instances | # Flip-Flops | # Bigrams |
|---|---|---|---|
| jpeg | 40K | 4K | 60K |
| dec_viterbi | 61K | 26K | 200K |
| netcard | 303K | 66K | 856K |
| leon3mp | 450K | 100K | 1.8M |
| megaboom | 990K | 350K | 3.4M |
| artificial | 2.4M | 400K | 1.3M |

We conduct three experiments to demonstrate accuracy and robustness of our predictive model. We also propose another experiment to generate endpoint-consistent PBA-GBA divergence.

- **Experiment 1. (Accuracy)**: The goal of this experiment is to validate our modeling accuracy. Model is trained with 70% data points of a real design and tested on unseen 30% data points of the same design.
- **Experiment 2. (Robustness)**: The goal of this experiment is to validate our modeling robustness. Model is trained with data points from post-CTS database of a real design and

tested on unseen post-routed implementation of the same design.

- **Experiment 3. (Robustness)**: The goal of this experiment is to validate the span of our artificial testcase development methodology. Model is trained with artificially generated testcases along with a sample of data points (30%) from a real design and tested on unseen 70% data points of the same real design.

- **Experiment 4. (Endpoint Slack)**: The goal of this experiment is to translate path-consistent PBA-GBA divergence predictions to endpoint-consistent PBA-GBA divergence values.

### 2.5.2 Results

In our results, we first compare model predicted PBA arrival time values with reference PBA results using a commercial timer, while maintaining path consistency. Reduction of model prediction divergence metrics as compared to reference divergence metrics signify the reduction of PBA-GBA divergence and availability of timing slack for design optimization.

**Results of Experiment 1:** We use 70% of the timing paths for training and test on 30% of the timing paths of the same design. Figure 2.10 illustrates the results for Experiment 1. As described in Table 2.6, mean, $99^{th}$ percentile and max divergence metrics reduce by at least 61.7%, 15.9% and 47.5%, respectively as compared to reference divergence metrics.

**Table 2.6.** Model divergence improvement in Experiment 1.

| | Mean | | 99p | | Max | |
|---|---|---|---|---|---|---|
| Design | actual | model | actual | model | actual | model |
| megaboom | 2.59ps | 0.99ps | 43.95ps | 23.05ps | 110ps | 78ps |
| leon3mp | 10.55ps | 2.14ps | 30.29ps | 7.45ps | 50.78ps | 42.70ps |
| netcard | 6.70ps | 1.52ps | 22.62ps | 8.52ps | 39.59ps | 19.90ps |

**Results of Experiment 2:** We use timing report from post-CTS database as input for training and test the model on post-routed database of the same design. This model is particularly helpful to set realistic optimization criterion during routing. Figure 2.11 illustrates the results for Experiment 2. As explained in Table 2.7, mean, 99$^{th}$ percentile and max model divergence metrics reduce by at least 26.6%, 11.7% and 26.3%, respectively as compared to reference divergence metrics.

**Table 2.7.** Model divergence improvement in Experiment 2.

| | Mean | | 99p | | Max | |
|---|---|---|---|---|---|---|
| Design | actual | model | actual | model | actual | model |
| megaboom | 4.51ps | 3.31ps | 53ps | 36.62ps | 119.24ps | 89.65ps |
| leon3mp | 9.43ps | 6.06ps | 26.79ps | 19.72ps | 50.78ps | 39.46ps |
| netcard | 4.29ps | 2.49ps | 17.20ps | 9.78ps | 33.56ps | 29.63ps |

**Results of Experiment 3:** We use in-house developed artificial designs and a sample from real design (30% data points) for training and predict PBA-GBA divergence on the same real design (70% data points). This is an idealistic goal, where we develop artificial testcases that can span the entire space of real designs. Figure 2.12 illustrates results from experiment 3. As described in Table 2.8, mean, 99$^{th}$ percentile and max model divergence metrics reduce by at least 27.1%, 13.4% and 13.5%, respectively as compared to reference divergence metrics.

**Table 2.8.** Model divergence improvement in Experiment 3.

| | Mean | | 99p | | Max | |
|---|---|---|---|---|---|---|
| Design | actual | model | actual | model | actual | model |
| megaboom | 3.06ps | 2.23ps | 41.14ps | 31.04ps | 119.24ps | 103.21ps |
| leon3mp | 9.07ps | 4.50ps | 22.59ps | 19.55ps | 46.46ps | 33.96ps |
| netcard | 7.21ps | 2.78ps | 21.84ps | 9.58ps | 46.25ps | 31.24ps |

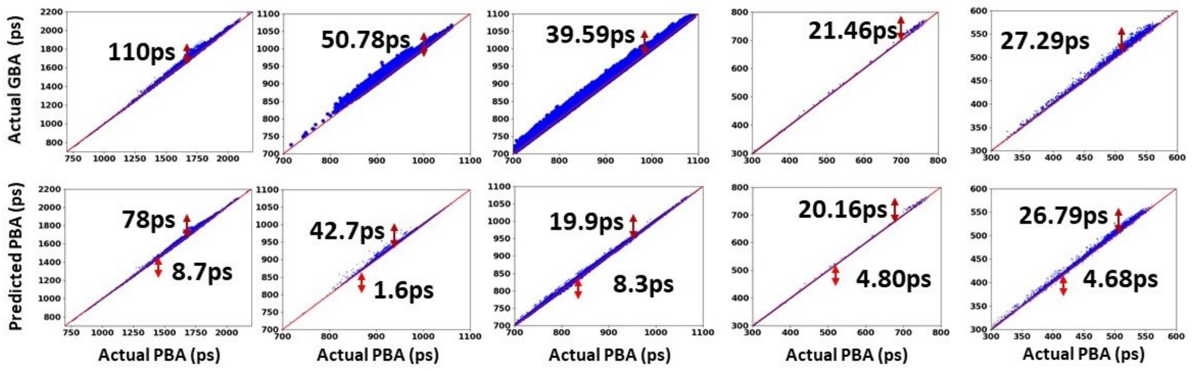


**Figure 2.10.** Results of Experiment 1 for actual GBA path arrival time (top row) and predicted PBA path arrival time (bottom row) versus actual PBA path arrival time for megaboom (left), leon3mp (middle) and netcard (right).

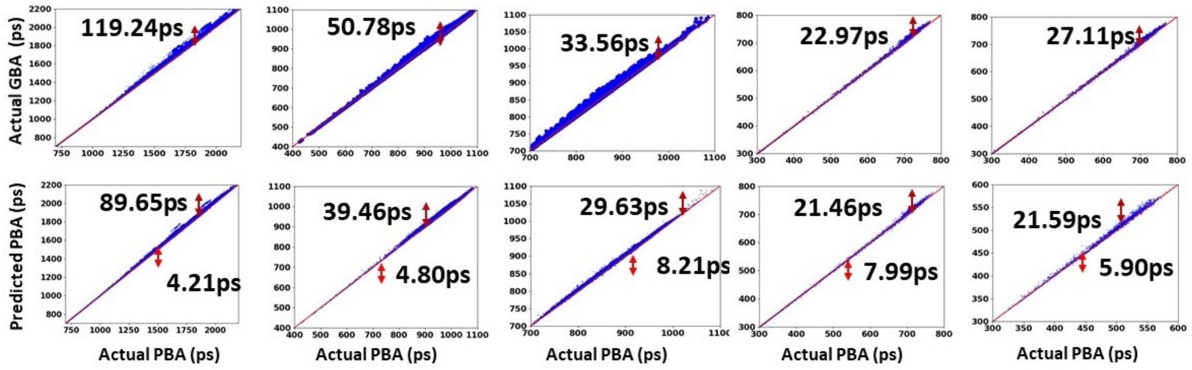


**Figure 2.11.** Results of Experiment 2 for actual GBA path arrival time (top row) and predicted PBA path arrival time (bottom row) versus actual PBA path arrival time for megaboom (left), leon3mp (middle) and netcard (right).

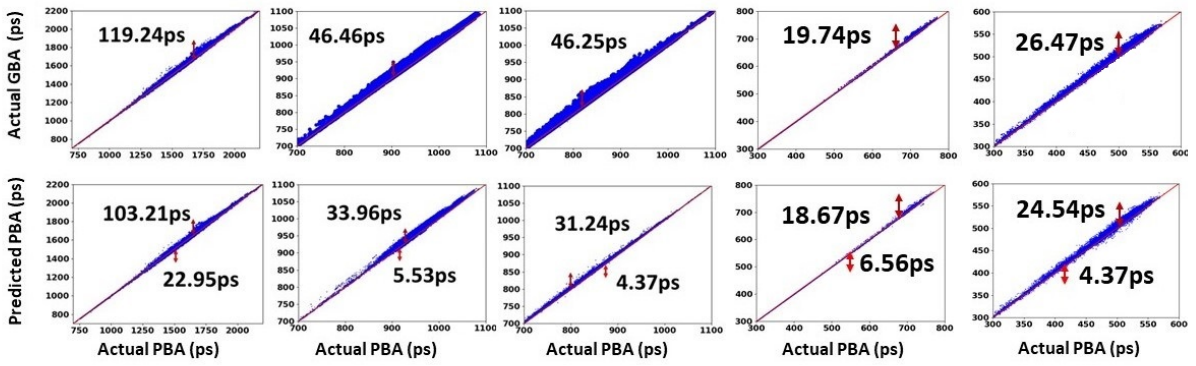


**Figure 2.12.** Results of Experiment 3 for actual GBA path arrival time (top row) and predicted PBA path arrival time (bottom row) versus actual PBA path arrival time for megaboom (left), leon3mp (middle) and netcard (right).

## 2.6 Unobserved Corner Prediction: Preliminaries

For the unobserved corner prediction task, Table 2.9 defines the notation that we use; see also Figure 2.13. We use $N$ to denote the total number of analysis corners and $n < N$ to denote the number of corners whose path delay values are *known*. We denote the set of known corners by $\{T_{known}\}$. Our goal is to accurately predict $N - n$ **unknown** path delay values from $n$ **known** path delay values. We use $\{T_{unknown}\}$ to denote the set of unknown corners. As seen in Figure 2.13, timing results can be viewed as elements in a large matrix whose rows represent timing paths and whose columns represent corners.[4] We use $R_{train}$ and $R_{test}$ to denote the numbers of timing paths that are used, respectively, in training and testing (i.e., inference) of our model. Thus, in the figure, the entire top portion of the matrix represents the *training data* that is used to train a predictive model. We refer to this matrix of training data as $M_1$.

A trained predictive model is used to infer the unknown elements in the bottom, *testing* (or, *inference*) matrix, which we refer to as $M_2$. Finally, we use $X_{train}$ and $X_{test}$ to respectively denote the matrix blocks of known corners that serve as inputs to our predictive model, and we use $Y_{train}$ and $Y_{test}$ to respectively denote the matrix blocks of unknown corners that our model is designed to predict.

**Table 2.9.** Terms and definitions for Unobserved Corner Prediction.

| **Term** | **Definition** |
|---|---|
| $N$ | Total number of columns in the matrix (= analysis corners) |
| $n$ | Number of known (i.e., analyzed) columns |
| $N-n$ | Number of unknown (i.e., to be predicted) columns |
| $\{T_{known}\}$ | Set of known columns |
| $\{T_{unknown}\}$ | Set of unknown columns |
| $R$ | Total number of rows in the matrix (= timing paths) |
| $R_{train}$ | Total number of complete rows (training) |
| $R_{test}$ | Total number of rows with missing columns (testing/inference) |
| $X_{train}$ | Matrix of known columns, used in model training |
| $X_{test}$ | Matrix of known columns, used in model testing/inference |
| $Y_{train}$ | Matrix of known columns, used in model training |
| $Y_{test}$ | Matrix of unknown columns, used in model testing/inference |
| $W$ | Model weight vector output by model training |
| $M_1$ | Matrix of size $R_{train} \times N$ |
| $M_2$ | Matrix of size $R_{test} \times N$ |
| $LRM$ | Linear regression model |

**Problem Statement.** We formally state our problem, which is a form of *matrix completion*, as follows.

**Given:** **(i)** a complete matrix $M_1$ split into $n$ column vectors $\{X_{train}\}_1^n$ and $N-n$ column vectors

[4]Our discussion will generally use the terms *corner* and *column* interchangeably. We also use both *testing* and *inference* to refer to evaluation of model accuracy on a set of unknown data.

$\{Y_{train}\}_n^N$, and containing $R_{train}$ rows of timing paths, along with **(ii)** an incomplete matrix $M_2$ split into $n$ known column vectors $\{X_{test}\}_1^n$ and $N-n$ unknown column vectors $\{Y_{test}\}_n^N$, containing $R_{test}$ rows of timing paths.

**Use:** $\{X_{train}\}_1^n$ and $\{Y_{train}\}_n^N$ to learn a model that can predict the unknown $N-n$ columns $\{Y_{test}\}_n^N$ of $M_2$ from the known $n$ columns $\{X_{test}\}_1^n$ of $M_2$.

This is a problem in multivariate regression. The simplest models for this purpose, which we report on in this paper, are linear regressors that attempt to minimize the mean squared error of predicted values (even as other metrics may be more meaningful for real-world evaluation). Experiments to evaluate and validate our models are described in Section 2.8.

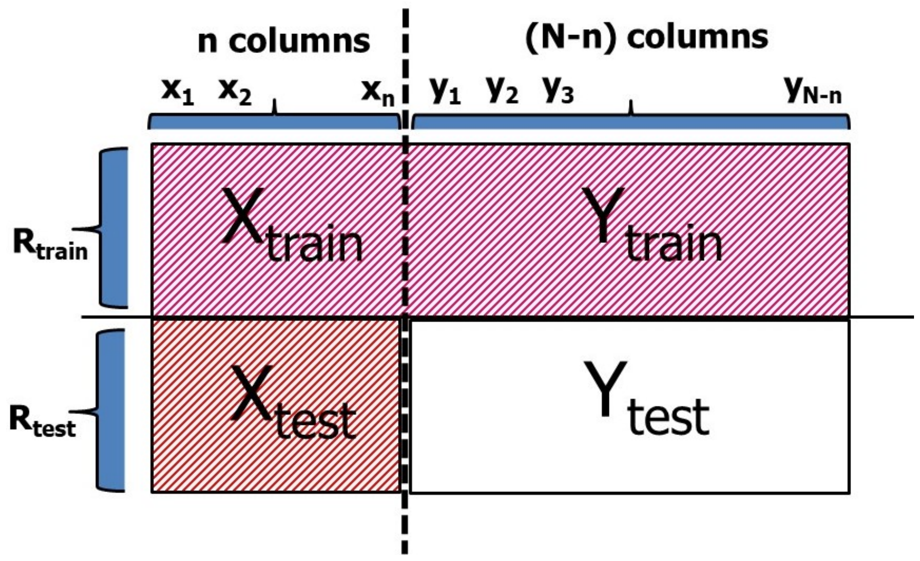


**Figure 2.13.** Visualization of timing prediction as a problem in matrix completion.

## 2.6.1 Intuition for Multivariate Linear Regression

Our overall approach is based on the premise that the path delay values at different analysis corners are strongly correlated. Intuitively, such correlations are a consequence of the underlying physics of devices, interconnects, and signal delay propagation along timing paths. Though it may be difficult to model the detailed physics that produce these correlations, we can measure and exploit these correlations to predict large numbers of unknown path delay values from smaller numbers of known ones. A useful exercise is to imagine the path delay values at different analysis corners as the coordinates of points in a high-dimensional space. Repeated analyses of these path values generate empirical distributions over this space; when we say that these values are strongly correlated, we mean that these distributions of points are far from uniform. Indeed, as we show later, the main support of these distributions is heavily concentrated in a much lower dimensional subspace. An equivalent observation is that the large data matrix, shown in Fig 2.13, can be very well approximated by a matrix of much lower rank: i.e., many of the columns of this matrix are either linearly dependent or very nearly so. For this reason, we might expect even simple linear methods in multivariate regression to excel at the task of predicting certain columns from collections of others. This is the hypothesis we investigate in this paper.

## 2.7 Unobserved Corner Prediction: Modeling Methodology

In this section, we describe our procedure for model definition, and our modeling flow. Our modeling procedure has three phases: subset selection, training, and testing.

**(i) Subset Selection.** In the context of our application, for a fixed value of $n$, the problem of *subset selection* is to determine which $n$ corners are most predictive of the remaining $N-n$ corners. For even moderately large values of $n$ and $N$, the number of possible subsets is prohibitively large to perform an exhaustive search. A common approach is to adopt a greedy strategy [66]; for our problem we use the simple strategy of *greedy deletion* (also known as *backward elimination* [67]).[5] The outcome of this procedure is an array sequence [*predict_corners*] of length $N-1$ whose order tells us (for any value of $n$) which $n$ corners should be used to predict the remaining $N-n$ corners.

[5]Our separate studies indicate that greedy *addition* results in worse model quality, especially for the small values of $n$ that are of interest.

**(ii) Model Training.** Once $n$ and corresponding $\{T_{known}\}$ are determined, the *training* phase finds model parameters $W^*$ for each $n$, such that the mean squared error of predictions is minimum. This is a one-time training investment.

$$W^* = \underset{W}{\operatorname{argmin}} ||Y_{train} - X_{train} * W||_2^2$$

**(iii) Model Testing/Inference.** The training phase generates optimal model parameters $W^*$ for each value of $n$. Each model parameter set $W^*$ has an associated error value. The cardinality of $\{T_{known}\}$ is chosen using an error budget that the user finds tolerable. The subset $\{T_{known}\}$ is then derived from the $[predict\_corners]$ array found by greedy deletion. Using the corresponding $W^*$, the model predicts timing at unobserved corners $\{T_{unknown}\}$ from observed corners $\{T_{known}\}$.

$$Y_{test} = X_{test} * W^*$$

**Modeling Flow.** We study the modeling flow illustrated in Figure 2.14. As we have noted, we first determine optimal model parameters $W^*$ for each value of $n$. The Inference phase predicts timing results of a test matrix with $n$ known columns. Use cases for our model are elaborated in Section 2.8. An important capability in practice will be to incrementally train models by including outliers found from inference results, e.g., at every $k$ executions of the model inference phase. This would allow the model to learn from outliers, such that subsequent mispredictions can be contained. Such a methodology might follow the feedback loop (blue) indicated in Figure 2.14.[6]

[6]In a typical SOC physical implementation methodology, a given block (hard macro) may go through SP&R&Opt steps many times, over the course of several months, each time with slightly different constraints or floorplan. We believe that our modeling approach can naturally exploit such a design process context and timeline.

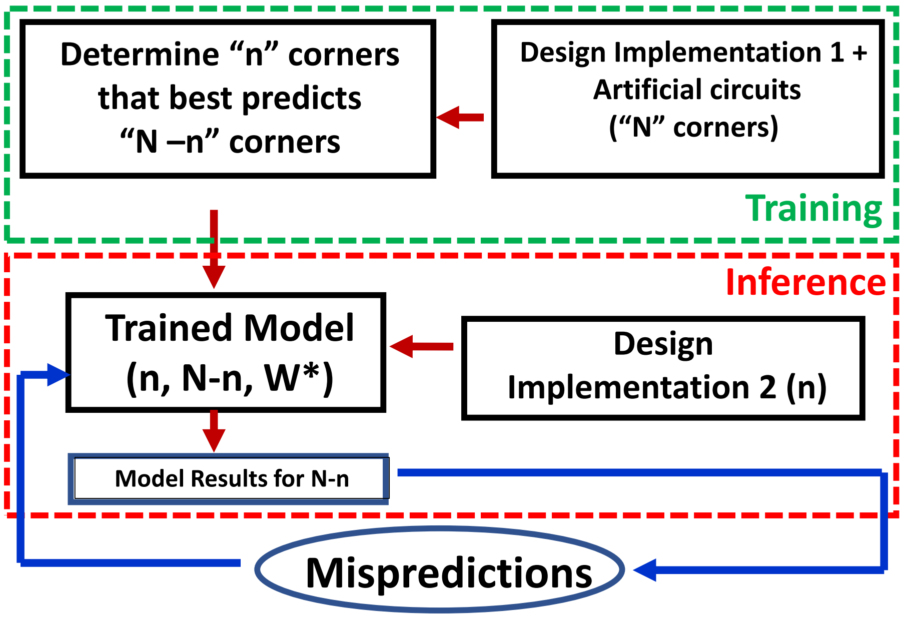

Determine "n" corners that best predicts "N –n" corners
Design Implementation 1 + Artificial circuits ("N" corners)
Training
Inference
Trained Model (n, N-n, W*)
Design Implementation 2 (n)
Model Results for N-n
Mispredictions

**Figure 2.14.** Our modeling flow. The potential feedback loop (blue arrows) is discussed in Sections VI below.

### 2.7.1 Data Generation

Recall that timing results are represented using matrices that can be fed into our models. We now describe our data generation, including design data and analysis conditions used in our experiments, followed by flows for our artificial circuit generation and matrix generation.[7]

Table 2.10 describes the data used in our modeling experiments. We use four public designs obtained from [46] [47] [48] along with in-house developed artificial circuits. We also evaluate our model on timing path data from three industrial designs (*prod1*, *prod2* and *prod3*) in sub-14nm technology nodes. Table 2.11 describes the space of analysis corners for our experiments in 28nm FDSOI (Experiments 1-4 below); an analogous space is used for 16nm experiments, and we have only limited insight into the sub-14nm test data obtained from our industry collaborator.

**Artificial Circuits Generation.** We have evaluated the potential benefit of artificial circuits (i.e.,

[7]We do not separately present data preparation for inference. Reprising previous discussion, the data flow for model inference is as follows. Recall that some number of known corners $n$ has been determined. Given $n$, we generate the test matrix, $X_{test}$, from timing results at $n$ known corners. Then, using the trained model parameters $W^*$ for the $n = |\{T_{known}\}|$ corners, we predict timing results in unknown corners $\{T_{unknown}\}$ of the test matrix.

**Table 2.10.** Design data used for experiments.

| Design | # Instances | # Flip-Flops | # Data points |
|---|---|---|---|
| *dec_viterbi* | 61K | 26K | 168K |
| *netcard* | 303K | 66K | 186K |
| *leon3mp* | 450K | 100K | 744K |
| *megaboom* | 990K | 350K | 510K |
| *artificial* | 2.4M | 400K | 746K |
| *prod1* | - | - | 21K |
| *prod2* | - | - | 111K |
| *prod3* | - | - | 27K |

**Table 2.11.** Space of 28nm FDSOI analysis corners (Expts 1-4).

| Parameter | Values |
|---|---|
| Process | SS, TT, FF |
| Voltage (V) | 0.6, 0.65, 0.70, 0.75, 0.80, 0.85, 0.90, 0.95, 1.00, 1.05, 1.10, 1.15, 1.30 |
| Temperature (C) | -40, 125 |
| BEOL corner | RCWORST |

small timing paths) used during an initial, “bootstrap” training phase of modeling. Algorithm 1 describes our artificial circuit generation flow. Input to this flow is a configuration file that contains circuit variables such as the number of stages in a timing path, $\{num\_stages\}$; standard cell types in the path defined by $\{Cell1\}$, $\{Cell2\}$, $\{Cell3\}$ and $\{Sink\}$; launch and capture flop-types defined by $\{LFlop\}$ and $\{CFlop\}$; aggressor cell types defined by $\{Agg\_Cell\}$; load cap range defined by $CL_{range}$; transition (slew) time values defined by $TR_{range}$; and clock period values defined by $\{Period\}$. The circuit generation sweeps through defined values for each variable; random values are generated between 0 and $CL_{range}$, and between 0 and $TR_{range}$, for these two variables. For each combination of the above-defined configuration variables, our flow generates a gate-level netlist (doe.v) (which comprises a collection of paths), along with an associated parasitic file (doe.spef), a transition annotation file (doe.timing) and a constraints file (doe.sdc).

Figure 2.15 shows a schematic of a timing path in our artificial netlist. The path has three stages between a launch flop and a capture flop. Our circuits capture a wide range of coupling

**Algorithm 1.** Artificial circuit generation.

```
Input: Configuration file containing $\{num\_stages\}$, $\{Cell1\}$, $\{Cell2\}$, $\{Cell3\}$, $\{Sink\}$, $\{LFlop\}$,
$\{CFlop\}$, $\{Agg\_Cell\}$, $CL_{range}$, $TR_{range}$, $\{Period\}$
Output: Design data for timing analysis
$Sol = \{doe.v, doe.spef, doe.sdc, doe.timing\}$
for $i$ in $\{num\_stages\}$ do
    for $j$ in $\{LFlop\}$ do
        for $k$ in $\{CFlop\}$ do
            for $c1$ in $\{Cell1\}$ do
                for $c2$ in $\{Cell2\}$ do
                    for $c3$ in $\{Cell3\}$ do
                        for $s1$ in $\{Sink\}$ do
                            for $p1$ in $\{Period\}$ do
                                doe.v ← genVerilog () // netlist generation
                                doe.spef ← genSpef () // spef generation
                                doe.timing ← genTiming () // slew annotation
                                doe.sdc ← genSDC () // constraints generation
                            end for
                        end for
                    end for
                end for
            end for
        end for
    end for
end for
$Sol \leftarrow \{doe.v, doe.spef, doe.sdc, doe.timing\}$
return $Sol$ =0
```

capacitance, wire capacitance and input transition values along with various combinations of standard cells.

**Matrix Generation Flow.** Once we have design data and corresponding timing graphs for various analysis corners, we translate these timing results into an equivalent matrix in which rows represent delay values of timing paths (i.e., a single, fixed path per row), and columns represent corners. In anticipated usage, this matrix would be input to model training.

We have considered two methods of matrix generation.

(1) Our *Endpoint* method finds $\{P_{endpoint}\}$ timing paths covering the worst timing paths across all endpoints in one corner. It then evaluates timing for these $\{P_{endpoint}\}$ paths in the rest of the $N-1$ corners. This ensures that all endpoints are covered, but the collection of paths can be

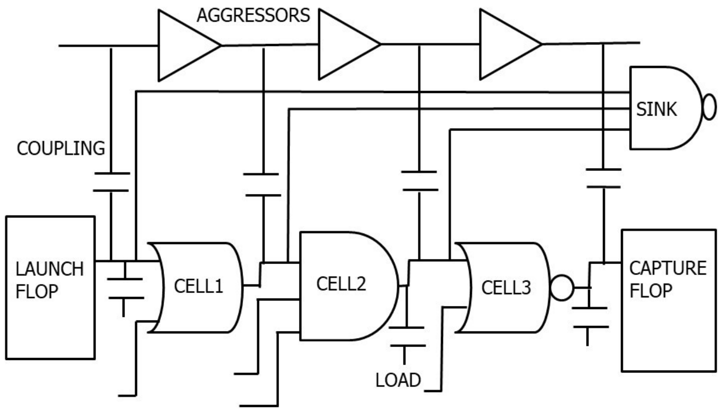


**Figure 2.15.** Illustration of artificial circuits.

biased toward worst paths of a single corner. Algorithm 2 details the matrix generation flow using the *Endpoint* method. Inputs are timing database sessions at *N* corners; the flow generates an $\{R_{train} \times N\}$ matrix with path delay values.

(2) Our *Union* method finds $\{P_i\}$ timing paths (one for each endpoint) from each corner's timing session $db_i$. It then collects the union of timing paths $\{P_{union}\}$ across all corners. Timing is evaluated for each unique path in $\{P_{union}\}$, in all corners. Since the cardinality of $\{P_{union}\}$ is high, we restrict the number of endpoints to a tractable number, *num*. This approach avoids the corner bias of the *Endpoint* method and ensures that the generated matrix is richer in terms of coverage of timing paths. Algorithm 3 details the matrix generation flow using the *Union* method. Again, inputs to this flow are timing database sessions at *N* corners, and an $\{R_{train} \times N\}$ matrix with path delay values is produced. We have evaluated both methods and observe only negligible differences in our inference results. Since the *Union* method is general and is less susceptible to corner bias, we use the *Union* method for all experimental validations reported below.

## 2.8 Unobserved Corner Prediction: Experimental Validation

As noted in the introduction, our simple, regression-based modeling approach is premised on the fact that the path delay values at different analysis corners are strongly correlated. One

**Algorithm 2.** Matrix generation by *Endpoint method.*

```
Input: Timing database sessions at $N$ corners $DB = \{db_1, db_2, ..., db_N\}$
Output: Matrix with path delay values
        $Sol$ = Matrix $M_1$ $\{R_{train} \times N\}$ =0
1: $\{P_{endpoint}\} \leftarrow$ getTimingPaths($db_1$, $nworst = 1$)
   // list of timing paths in corner1.
2: for $j$ in $DB$ do
3:     for $k$ in $\{P_{endpoint}\}$ do
4:         $M_1[k][j] \leftarrow$ evalTiming($j$, $k$)
           // estimate delay for $k^{th}$ path in $db_j$
5:     end for
6: end for
7: $Sol \leftarrow M_1$
   // Delay matrix $R_{train} \times N$.
8: return $Sol$ =0
```

way to exhibit these correlations is to perform a principal component analysis [68] of the data. Figure 2.16 plots the eigenvalues (normalized by the leading eigenvalue) of the covariance matrices for the data sets of the first four public benchmark designs listed in Table II. The relative magnitudes of these eigenvalues measure the relative variance captured by different principal components of the data. Note that while the data for each design consists of path delay values at 44 different analysis corners, the variance of the data is concentrated in a much lower dimensional subspace. In particular, several orders of magnitude separate the leading eigenvalues in these covariance matrices from those at the middle or bottom of the spectrum.

In the remainder of this section, we first describe our reporting metrics, and then describe the design of experiments to validate our predictive modeling approach. For each of these experiments, we discuss our modeling results.

### 2.8.1 Reporting Metrics

During the inference phase, we predict elements in the $Y_{test}$ region of Figure 2.13. This region has $R_{test}$ data points and $N - n$ columns. For each such element ($i^{th}$ row and $j^{th}$ column) of $Y_{test}$, we define ${d_m}^{ij}$ as the predicted path delay and ${d_a}^{ij}$ as the corresponding actual path delay from golden timing analysis. We also define ${\varepsilon_{abs}}^{ij}$ as the absolute error for each element of $Y_{test}$.

**Algorithm 3.** Matrix generation by *Union method.*

```
Input: Timing database sessions at $N$ corners $DB = \{db_1, db_2, ..., db_N\}$, number of endpoints $num$
Output: Matrix with path delay values
        $Sol$ = Matrix $M_1$ $\{R_{train} \times N\}$ =0
1:  for $i$ in $DB$ do
2:      $\{P_i\} \leftarrow$ getTimingPaths($i$, $num$, $nworst = 1$)
        // List of timing paths, $num$ endpoints per endpoint
3:  end for
4:  if True then
5:      $\{P_{union}\} \leftarrow \{P_1\}\ U\ \{P_2\}\ U\ \{P_3\}$ .. $U\ \{P_N\}$
        // union of paths across all corners
6:      $\{P_{unique}\} \leftarrow$ uniquePaths($\{P_{union}\}$)
        // unique paths from union of timing paths
7:  end if
8:  for $i$ in $DB$ do
9:      for $j$ in $\{P_{unique}\}$ do
10:         $M_1[j][i] \leftarrow$ evalTiming($j$, $i$)
            // estimate delay for path $j$ in $i$
11:     end for
12: end for
13: $Sol \leftarrow M_1$
    // Delay matrix $R_{train} \times N$
14: return $Sol$ =0
```

Though our model aims at reducing the square of absolute mispredictions, we understand from industry collaborators that relative delay prediction error is a valuable criterion. Intuitively, a timing path with larger path delay value can afford larger misprediction, compared to a timing path with smaller path delay. This is because the former has scope to fix violations with data path optimizations, even with a misprediction; the latter, having less implementation flexibility, cannot afford misprediction as easily (hence, cost of model misprediction is higher). With this in mind, we also report relative errors ${\varepsilon_{rel}}^{ij}$.

To quantify the model accuracy for the entire $Y_{test}$ region of the matrix using a single value, we report three lumped metrics, namely, *mean*, $99^{th}$percentile and $99.99^{th}$percentile values denoted by $\varepsilon_{mse}$, $\varepsilon_{99p}$ and $\varepsilon_{99p99}$. Table 2.12 explains our model accuracy metrics.

## 2.8.2 Design of Experiments

In our experiments, we use in-house developed artificial designs and four public benchmark designs, listed in Table 2.10. We use 28nm FDSOI and 16nm foundry enablements for our model evaluation. We also use sub-14nm foundry enablement on three industrial designs for our model evaluation.

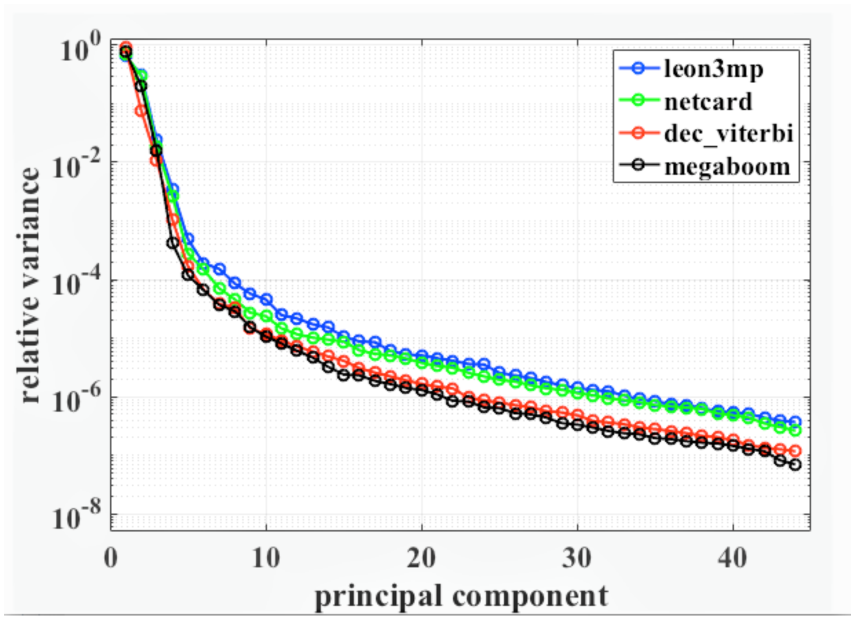


**Figure 2.16.** Relative variance captured by successive principal components of the first four data sets in Table 2.10 (44 analysis corners). In each data set, the variance is concentrated in a subspace of much lower dimensionality than the total number of analysis corners.

For the inference phase, we study two possible testing conditions. (i) *Matched testing* conditions hold when the data points in the inference phase match very closely to the data points of the training phase in their feature space. Such a condition implies structural similarity of timing paths in training and test data points. (ii) *Mismatched testing* conditions hold when data points in the training phase do not necessarily span the data points from the inference phase.

We demonstrate our model usage for both data and clock path delay predictions. For data path delay predictions, we conduct Experiments 1 and 2 in *Mismatched testing* conditions. For clock path delay prediction, we conduct Experiment 3 in *Matched testing* conditions, to demonstrate our model's applicability in predicting clock insertion delay. We conduct Experiment 4 in *Matched testing* conditions to assess whether the number of corners required to accurately predict unknown corners increases with growth of the total number of timing corners. This experiment demonstrates the scalability of our model. We propose Experiments 5 and 6 in *Matched testing* conditions to demonstrate our model usage in advanced technology nodes (16nm and sub-14nm foundry enablements, respectively).

**Table 2.12.** Model accuracy metrics.

| **Notation** | **Meaning** |
|---|---|
| ${d_m}^{ij}$ | Model predicted delay ($i^{th}$ row and $j^{th}$ column of $Y_{test}$) |
| ${d_a}^{ij}$ | Actual delay corresponding to $i^{th}$ row and $j^{th}$ column of $Y_{test}$ |
| ${\varepsilon_{abs}}^{ij}$ | $\lvert {d_m}^{ij} - {d_a}^{ij} \rvert$ (Absolute error) |
| ${\varepsilon_{rel}}^{ij}$ | $\frac{{\varepsilon_{abs}}^{ij}}{{d_a}^{ij}}$ (Absolute relative error) |
| $\varepsilon_{mse}$ | $\sqrt{\frac{1}{R_{test}} \sum_{ij} ({\varepsilon_{rel}}^{ij})^2}$ (Root of mean squared relative errors) |
| $\varepsilon_w$ | $\max_{ij}\{\varepsilon_{rel}\}$ (Max of all absolute relative error values) |
| $\varepsilon_{99p}$ | $99^{th}$ percentile value of $\{\varepsilon_{rel}\}$ (when sorted in ascending order) |
| $\varepsilon_{99p99}$ | $99.99^{th}$ percentile value of $\{\varepsilon_{rel}\}$ (when sorted in ascending order) |

**Data Path Delay Model: Experiment 1.** The goal of this experiment is to validate our model's usefulness in *Mismatched testing* conditions. The model is trained with data points from post-routed implementation of a real design and tested on an unseen post-routed implementation of the same design. Since the timing paths are from two different physical implementations, we include this use case in *Mismatched testing* conditions. This use case is relevant to exploring multiple physical implementations of the same design (or, re-analyzing implementations through the months-long physical design process) without having to analyze timing in all corners.

**Data Path Delay Model: Experiment 2.** The goal of this experiment is to assess potential benefits of our artificial testcase development methodology. This reflects a hypothetical "ideal scenario", wherein we have the capability to produce artificial testcases that span the entire space of real designs. Though the intent of artificial circuits is to eventually be able to span the entire space of (likely) real designs, the current state of our artificial circuits can be treated as *Mismatched testing* condition since we observe real timing structures poorly covered by our artificial circuits. In this experiment, our model is trained with artificial testcases and tested on unseen real designs. The motivating scenario: a one-time trained model using artificial circuits can predict timing analysis in unknown corners of any real design, using timing analysis in few known corners of the same real design.

**Clock Insertion Delay Model: Experiment 3.** Accurate clock network synthesis and optimization are essential for advanced-node IC design. The goal of this experiment is to demonstrate the usefulness of our model in predicting clock insertion delay at unknown corners $\{T_{unknown}\}$ using clock insertion delay at known corners $\{T_{known}\}$. This ensures that the clock network is synthesized considering its delay values at all timing corners, without overdesigning the clock tree or leaving violations unattended till very late in the design cycle.

**Corner Scalability: Experiment 4.** In Experiments 1-3, we use $N = 44$ corners, and results (see next Subsection) indicate that a small subset of corners can accurately predict the rest of the unknown corners. Since design methodologies in advanced nodes require timing analysis at $N \gg 44$ corners, this experiment aims to discover whether increasing the number of corners $N$ demands an increase of $n$ on the same scale. We increase the number of corners from 44 to 82 and use *Matched testing* conditions to perform this experiment.

**Technology Independence: Experiments 5 and 6.** All of Experiments 1-4 use 28nm foundry enablement. Experiments 5 and 6 seek to confirm the utility of our modeling in more advanced technology nodes, specifically, 16nm and sub-14nm foundry enablements. We use *Matched testing* conditions for our validation.

### 2.8.3 Experimental Results

In all of our results, shown in the plots below, we report (y-axis) values of *mean*, $99^{th}$*percentile* and $99.99^{th}$*percentile* metrics, denoted as $\varepsilon_{mse}$, $\varepsilon_{99p}$ and $\varepsilon_{99p99}$ (see Table 2.12). On the x-axis of each plot, we show $n$ (number of known corners), ranging from 1 to $N-1$. Thus, for example, $n = 4$ on the x-axis of any given plot indicates that the model is predicting $N-4$ corners using $n = 4$ observed corners. Throughout our experimental results, it is evident that error metric generally reduces as more corners are included in $\{T_{known}\}$.

To recap analysis corners and technology enablements in our experiments: (i) we use $N = 44$ corners for Experiments 1, 2, 3 and 5, and $N = 82$ corners for Experiment 4; (ii) we use 28nm FDSOI libraries for Experiments 1, 2, 3 and 4, and 16nm libraries for Experiment

5; and (iii) for Experiment 6, we use three industrial designs with $N = 42$, $N = 58$ and $N = 29$, respectively, in sub-14nm foundry enablement.

**Results of Experiment 1.** For this experiment, we use timing paths from a physical implementation (0.85 utilization, aspect ratio 1) for training, and test on an unseen physical implementation (0.75 utilization, aspect ratio 0.9) of the same design. The plots in Row 1 of Figure 2.17 show that $\varepsilon_{mse} \leq 0.005$ (0.5% error) for $n = 4$, $\varepsilon_{99p} \leq 0.01$ (1% error) for $n = 5$ and $\varepsilon_{99p99} \leq 0.01$ (1% error) for $n = 14$.

**Results of Experiment 2.** For this experiment, we use timing paths from in-house developed artificial designs for training, and test on unseen real designs. The plots in Row 2 of Figure 2.17 show that $\varepsilon_{mse} \leq 0.01$ for $n = 18$, $\varepsilon_{99p} \leq 0.01$ for $n = 23$ and $\varepsilon_{99p99} \leq 0.01$ for $n = 32$. We believe that the non-monotonicity of the curves is a consequence of the *Mismatched testing* conditions. Also, the larger $n$ value to predict timing at unobserved corners also suggests the need for improvement of our artificial circuit generation methodology.

**Results of Experiment 3.** We use 10% of clock paths for training, and test on unseen 90% clock paths of the same design. Such a use case can bring the best of both accuracy and runtime worlds during synthesis and optimization of the clock network. The plots in Row 3 of Figure 2.17 show that $\varepsilon_{mse} \leq 0.005$ for $n = 3$, $\varepsilon_{99p} \leq 0.01$ for $n = 4$ and $\varepsilon_{99p99} \leq 0.01$ for $n = 6$.

**Results of Experiment 4.** We use 10% of data paths for training and test on unseen 90% data paths of the same design. The plots in Row 4 of Figure 2.17 show that $\varepsilon_{mse} \leq 0.005$ for $n = 4$, $\varepsilon_{99p} \leq 0.01$ for $n = 6$ and $\varepsilon_{99p99} \leq 0.01$ for $n = 23$. The fact that small values of $n$ achieve good model accuracy supports our initial hypothesis that the distribution of delays across different corners is highly concentrated in a low-dimensional subspace. Further, these results suggest that a small number of known analysis corners $n$ can suffice to accurately predict timing analyses at unknown corners, even as $N$ grows large.

**Results of Experiment 5 and 6.** We use 10% of data points from a real design, for training and test on unseen 90% data points of the same design. We use $N = 58$ for experiments using 16nm foundry enablement. The plots in Row 5 of Figure 2.17 show that $\varepsilon_{mse} \leq 0.005$ for

$n = 11$, $\varepsilon_{99p} \leq 0.01$ for $n = 6$ and $\varepsilon_{99p99} \leq 0.01$ for $n = 21$.

Figure 3.11 shows results using sub-14nm technology libraries on industrial designs *prod1*, *prod2* and *prod3*. The plots in Figure 3.11 show that $\varepsilon_{mse} \leq 0.005$ for $n = 5$, $n = 7$ and $n = 9$ for designs *prod1*, *prod2* and *prod3* respectively. $\varepsilon_{99p} \leq 0.02$ for $n = 6$, $n = 9$ and $n = 12$ for designs *prod1*, *prod2* and *prod3* respectively. $\varepsilon_{99p99} \leq 0.03$ for $n = 8$, $n = 21$ and $n = 15$ for designs *prod1*, *prod2* and *prod3* respectively.

**Worst-case Errors and Outliers.** The results of Experiment 6 show that improved accuracy in sub-14nm nodes is an important direction for our future work. Furthermore, in each of our experiments, we see a handful of data points that fail to be reconstructed to their high-dimensional space accurately, even with large values of $n$. (In the industry datasets studied with Experiment 6, we understand that outliers are unsurprising for reasons such as (i) existence rare path types (e.g., memory as opposed to reg-to-reg) with limited training examples, and (ii) existence of isolated corners with no similar corners in the provided dataset.) While root-cause analysis and improvement of outlier (high $\varepsilon_w$) predictions is another important direction for our future work, we note that industry design methodology teams consider such outliers to be expected, and that effects of a few mispredictions are insignificant relative to (i) the analysis improvement and design convergence benefits from a predictive model, and (ii) analysis inaccuracies that exist in current methods.[8]

## 2.9 Conclusion

To address the accuracy-runtime [9] [49] tradeoff of STA engines, we apply machine learning techniques to model PBA-GBA divergence in endpoint arrival times. We propose a model based on decision trees along with electrical and physical parameters of *stage bigrams* in timing paths. We assess potential benefits of our model using 28nm FDSOI foundry technology,

[8]Our industry collaborator indicates that minor violations must be dealt with at the end of timing closure anyway, hence worst-case outliers even with 10% or greater prediction error would not cause concern. If deemed necessary, such outliers could be caught up front by an STA run covering all corners, incurring a one-time cost. Incremental model training, as suggested in the blue arrows of Figure 2, could also help cure outliers through iterations of the physical design process.

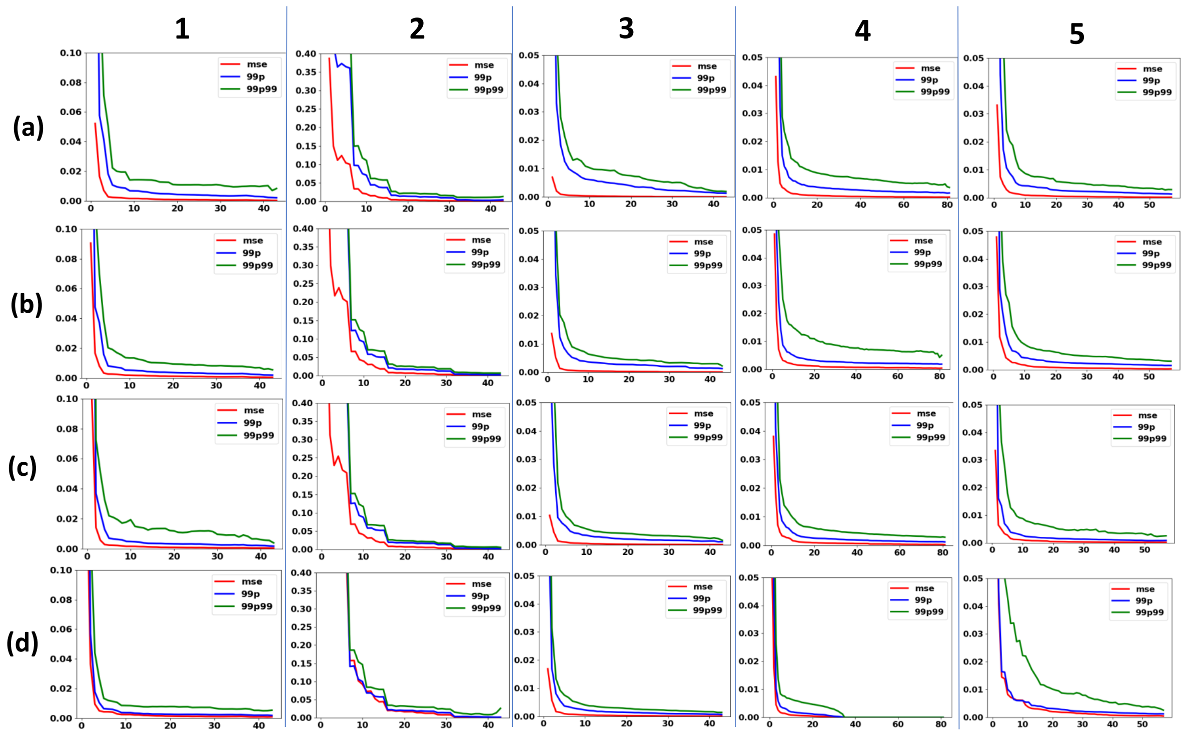


**Figure 2.17.** Results of Experiments 1, 2, 3, 4 and 5 (top to bottom): Plots of $\varepsilon_{mse}$ (red), $\varepsilon_{99p}$ (blue) and $\varepsilon_{99p99}$ (green) versus $n$, for designs *dec_viterbi*, *netcard*, *leon3mp* and *megaboom* (left to right).

a leading commercial signoff STA tool, and implementations of public testcase designs up to 1M+ instances. We measure the decrease of PBA-GBA divergence obtained by the model, according to several metrics and in several usage scenarios. In our experiments, model-predicted PBA arrival times reduce mean, $99^{th}$ percentile and max divergence metrics by at least 26.6%, 13.4% and 11.7%, respectively as compared to reference PBA-GBA divergence metrics. Such reductions can help avoid over-fixing and achieve improved power and area outcomes during optimization. In addition, both model training and inference are efficient, with a training time of 219 seconds and testing time of 17 seconds for a test dataset with 1.04M bigrams. A number of ongoing and future works remain. Chiefly, we are seeking to integrate our predictive models with an academic sizer and optimizer, to explore the benefit from reduced pessimism in MCMM timing closure and sizing for leakage and total power reduction. As shown by Experiment 3, significant work remains toward design of artificial testcases that can train well-performing models for a given

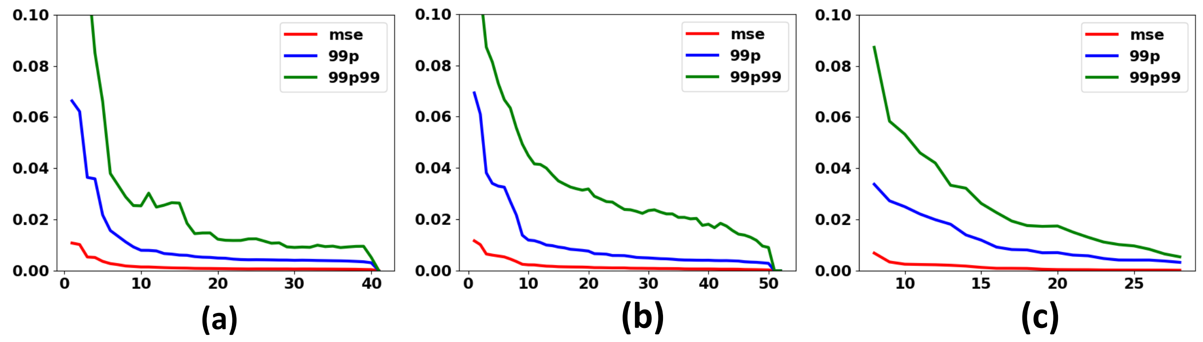


**Figure 2.18.** Results of Experiment 6 using sub-14nm technology libraries. Plots of $\varepsilon_{mse}$ (red), $\varepsilon_{99p}$ (blue) and $\varepsilon_{99p99}$ (green) versus $n$ for industrial designs (a) *prod1* (b) *prod2* and (c) *prod3*.

design enablement, independent of any actual designs. Reduction or elimination of remaining optimism in PBA slack prediction, as well as endpoint-consistent pessimism reduction, present additional challenges for future research.

A significant portion of the design cycle is spent on the static timing analysis (STA) at multiple corners and multiple modes (MCMM). For applying learning-based models for timing delay prediction at unobserved corners, we have taken a data-driven approach to model the physics of timing delays across multiple timing corners. Our approach is based on the premise that the path delay values at different analysis corners are strongly correlated. In particular, viewing these path delay values as the coordinates of points in a high-dimensional space, we have observed that the main support of their distribution is heavily concentrated in a much lower dimensional subspace. As a consequence, we have shown that simple linear methods in multivariate regression can accurately predict a large set of unknown corners from a smaller set of known ones. For example, with a 1M-instance example in foundry 16nm enablement (10% training, 90% testing), we obtain a model based on 10 observed corners that predicts timing results at the remaining 48 unobserved corners with less than 0.5% relative root mean squared error, and 99$^{th}$ percentile relative prediction error less than 0.6%. We are currently exploring numerous other directions to address further challenges. With large data sets, for example, it is possible to learn more flexible statistical models that do not make strong assumptions of linearity. Also, to handle outliers, it is possible to optimize more robust criteria in our statistical fits. So far these approaches have yielded incremental benefits, but it remains to find the optimal combination of strategies for the problem of timing analysis in advanced-node IC design. These and other issues are the subject of our ongoing and future work.

Chapter 2 contains material from “Using machine learning to predict path-based slack from graph-based timing analysis”, by Andrew B. Kahng, Uday Mallappa and Lawrence Saul, which appears in International Conference on Computer Design, October 2018; “Unobserved Corner Prediction: Reducing Timing Analysis Effort for Faster Design Convergence in Advanced-Node Design”, Andrew B. Kahng, Uday Mallappa, Lawrence Saul and Shangyuan Tong, which appears in Design, Automation and Test in Europe Conference & Exhibition, March 2019. The dissertation author was the primary investigator and author of these papers.

# Chapter 3

# AI for EDA: Design Optimization

The solution space of detailed placement becomes intractable with increase in thenumber of placeable cells and their possible locations. So, the existing works either focus on the sliding window-based optimization or row-based optimization. Though these region-based methods enable us to use linear-programming, pseudo-greedy or dynamic-programming algorithms, locally optimal solutions from these methods are globally sub-optimal with inherent heuristics. The heuristics such as the order in which we choose these local problems or size of each sliding window (runtime vs. optimality tradeoff) account for the degradation of solution quality. Our hypothesis is that learning-based techniques (with their richer representation ability) have shown a great success in problems with huge solution spaces, and can offer an alternative to the existing rudimentary heuristics. We propose a two-stage detailed-placement algorithm `RLPlace` that uses reinforcement learning (RL) for coarse re-arrangement and Satisfiability Modulo Theories (SMT) for fine-grain refinement. With global placement output of two critical IPs as the start point, `RLPlace` achieves upto 1.35% HPWL improvement as compared to the commercial tool's detailed-placement result. In addition, `RLPlace` shows at least 1.2% HPWL improvement over highly optimized detailed-placement variants of the two IPs.

## 3.1 Introduction

In the physical design flow, the detailed placement step follows the global placement step. The job of the detailed placer is to legalize the cells in valid placement sites. While doing this, several objective functions such as wirelength, power, timing, area, etc. are used to improve the solution quality. Since the design is already optimized for one of the above cost metrics before the detailed placement step, the cost improvement during the detailed placement is incremental and low in magnitude as compared to the optimization of global placement step; thus making it a difficult problem. As the number of placeable cells and their possible placement locations increase, heuristic-based solvers serve as a good alternative for the detailed placement problem. Region-wise optimization involves dividing the layout into multiple smaller and tractable regions and work on each of these small local regions in some sequence. The inherent heuristics such as order of selecting the regions plays an important role in the final solution quality. Our hypothesis is that Reinforcement Learning (RL) techniques can help us in determining the optimal sequence of choosing the regions for such region-based optimizations. The most important feature of a such a learning-based placer is its ability to learn from past experience and improve its performance over training. Besides the learning ability, there are also two other promising properties of learning-based placers. Firstly, the trial-and-error mechanism of RL makes it easier to deal with lack of data training data. Secondly, the machine learning models in general, tend to have a better state representation ability with flexible representation forms such as image (matrix), vector and graph, which is usually not available for heuristics-based methods. Recent works in RL for solving Combinatorial Optimization (CO) problems [69] combine RL with existing combinatorial search algorithms and demonstrate promising results. These techniques leverage deep RL's ability to recover reasonable rough solution fast, and then exploit the ability of exact solvers to incrementally improve via local refinement.

**Problem Statement:** Given a global placement solution, we divide the layout into multiple grids (a set of grids form a window or a region). The goal of sliding-window based

RLPlace algorithm is to determine optimal arrangement of grids in each window and distribute the cells within each grid such that the resulting overlap-free solution is optimal in terms of total wirelength.

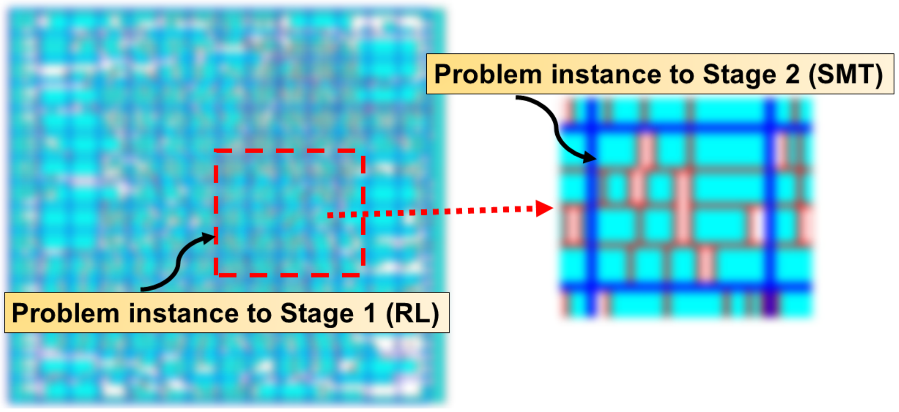

Problem instance to Stage 2 (SMT)
Problem instance to Stage 1 (RL)

**Figure 3.1.** The problem instances of RLPlace , shown on a blurred layout of our design.

## 3.2 Related Work

The proliferation of machine learning algorithms coalesced with GPU acceleration offer an alternative to solve existing EDA problems [70] with huge solution space, along with generalization to unseen problems. Many previous works use supervised learning techniques for various VLSI optimization problems; [71], [72] for prediction of congestion and DRC violations, [73, 74] for predictions associated with static timing analysis, [27] for power-delivery network synthesis and [29] for leakage power optimization. In active learning category, deep RL [69, 75] has proven its success in problems formulated as MDP. Previous works [76, 77, 78, 34, 79] demonstrate the efficacy of deep learning-driven prediction problems and reward-driven optimization tasks such as macro-placement, transistor sizing, tool parameter optimization. Recently, Lin et al. [80] propose batch-based concurrent algorithms ABCDPlace that exploits GPU acceleration.

**Our Contribution:** Though previous works use RL for macro-placement (piggybacked by forced directed standard-cell placement), we are the first to formulate the standard-cell detailed-placement as MDP and solve it using RL.

## 3.3 Placement as MDP

### 3.3.1 Markov Decision Process (MDP)

Optimization problems such as VLSI placement can be viewed as a sequence of decisions; decisions to gain some long-term reward such as wirelength, area, performance or power. Markov Decision Process (MDP) gives us a formal way to solve these sequential decision making processes. Formally, MDP is a discrete-time state-transition system that can be described with four components (i) States ($s \in S$) that are necessary for choosing possible actions (ii) Actions $a \in A$ that are chosen from a policy $\pi : S \to A$ (iii) Transition Model $P(s'|s,a)$ that describe the dynamics of the world and (iv) Reward that is a real-valued function $R(s)$ on states representing the goodness of a state, from the agent's perspective. The value of a state $V^{\pi}(s)$ keeps track of the expected long-term discounted rewards of the state. The discount factor $\gamma < 1$ guarantees the convergence of expected reward. Practically, this makes sense in the placement problem, because we are not certain of long-term future and is rational to weigh them less as compared to immediate rewards.

$$V^{\pi}(s_0) = \mathbb{E}_{\pi}\Big(\sum_{i=0}^{\infty} \gamma^i R(s)\Big) = \sum_{i=0}^{\infty} \gamma^i * R(s) \leq \frac{R_{max}}{1-\gamma} \tag{3.1}$$

### 3.3.2 Reinforcement Learning

The value function for every state has an interesting local recursive relationship, represented by "Bellman Expectation" Equation 3.2. The first term $R(s)$ is deterministic, given the state. The second term is a distribution over all possible next-state transition probabilities.

$$V^{\pi}(s) = R(s) + \gamma * \sum_{s'} P\big(s'|s, \pi(s)\big) V^{\pi}(s') \tag{3.2}$$

For every state, we are interested in the optimal value function $V^*(s)$ and then derive the corresponding policy.

$$V^*(s) = \max_{\pi} \big(V^{\pi}(s)\big) = R(s) + \max_{a \in A(s)} \left( \sum_{s'} P\big(s'|s, a\big) V^*(s') \right) \tag{3.3}$$

In the absence of transition probabilities of the environment, learning from simulated experiences a powerful method to determine the value function. The action-value states denoted by $Q$ states help us to overcome the sampling issue associated with the max operator of Equation 3.3. Intuitively, imagine taking an action $a$ and then following the default policy thereafter.

$$Q(s,a) = R(s,a) + \gamma \left( \sum_{s'} P\big(s'|s,a\big) \max_{a'} Q(s',a) \right) \tag{3.4}$$

We can use the Temporal Difference (TD) update rule of Equation 3.5, to incrementally improve the Q value estimates as we get more and more samples. Once the $Q$ values converge, the best value of a state can be obtained by Equation 3.6.

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left( R(s) + \gamma \max_{a'} Q(s',a') - Q(s,a) \right) \tag{3.5}$$

$$V^*(s) = \max_{a} Q(s,a) \tag{3.6}$$

## 3.3.3 Deep Q Learning

To derive the optimal policy, we can either maintain a table of $Q$ values for all state-action $(s,a)$ pairs or use a function approximator (using a neural network). For both the methods, the idea is to iteratively improve the stored current $Q$ values using the TD update. The obvious advantage of having a function approximator is the scalability (without having to deal with large tables), along with generalizability.

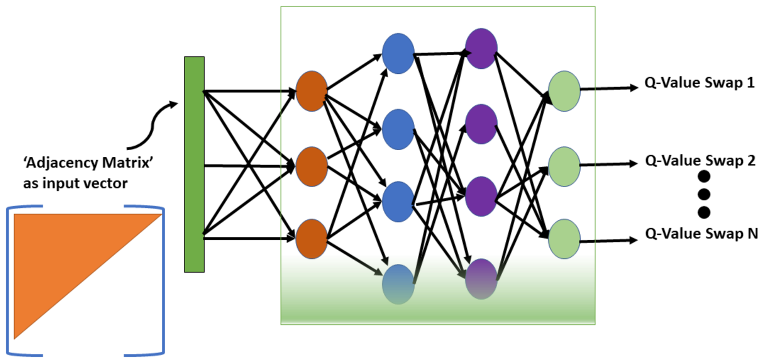


**Figure 3.2.** Neural network takes state of the window as the input and learns to distinguish good actions (grid swaps) against the bad actions.

## 3.3.4 Satisfiability Modulo Theories (SMT)

For optimization problems such as placement, in addition to satisfiability, we also need to support for predicate logic along with an ability to systematically search for the optimal solution. To solve such Boolean Optimization problems, Satisfiability Modulo Theories-based framework (SMT) often termed as the generalization of Boolean SAT solvers offer a richer modeling language and framework; AND, OR, at-most-1 (AMO), at-least-1 (ALO), exactly-1 (EO), if-then-else (ITE), BitVector (BV) representations and lexicographic objective functions. In our flow, once we determine optimal sequence of grid-swaps of each window, we use the SMT framework for fine-grain refinement.

## 3.4 RL for Detailed Placement

In this work, we follow a two-stage hierarchical approach to solve the problem of detailed placement optimization. We divide the layout into windows or regions (2D space); similar to the sliding window approach. Each window is further partitioned into grids that contain many cells. In Stage 1, instead of working at the cell-level, we work on rearranging these coarse grids that contain one or more cells. When we rearrange the grids inside each window, the cells associated with the grid move along with the grid. It is important to observe that any possible arrangement of the grids in the window can be obtained by a sequence of grid-swaps. Therefore, the eventual goal of Stage 1 is to determine the optimal sequence of these grid swaps and handover to Stage 2 to perform the fine-grain overlap-free redistribution. Figure 3.1 summarizes our RL model training and the two-stage `RLPlace` flow-deployment.

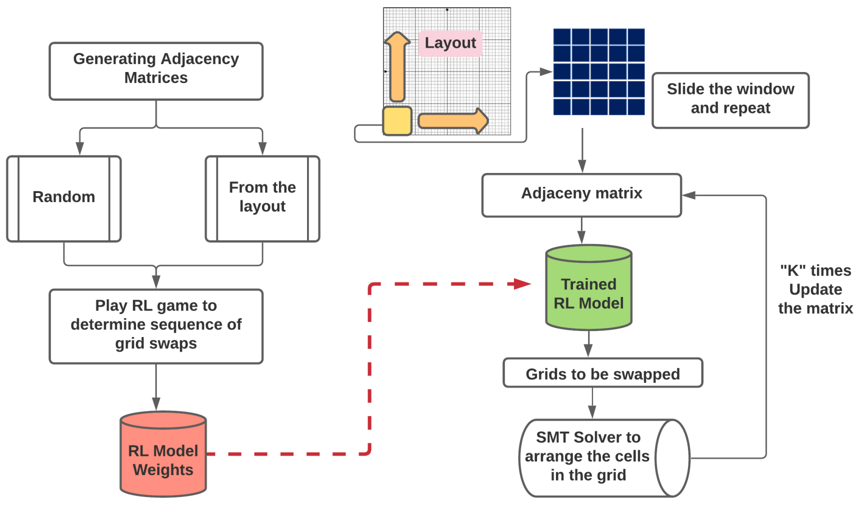

Generating Adjacency Matrices
Random
From the layout
Play RL game to determine sequence of grid swaps
RL Model Weights
Layout
Slide the window and repeat
Adjaceny matrix
Trained RL Model
"K" times Update the matrix
Grids to be swapped
SMT Solver to arrange the cells in the grid

**Figure 3.3.** The flow on the left shows the learning phase of our RL framework and the right is model deployment.

### 3.4.1 Stage 1 (coarse-grained placement)

We work on a window of the layout at a time, in the pursuit of generating an optimal grid arrangement in the window. Even a $5 \times 5$ window results in $25!$ possible grid arrangements, indicating the complexity of the problem. As this window is swept across the layout, the exercise of grid re-arrangement needs to be performed multiple times. We observe that any re-arrangement of grids in a window can be realized using a sequence of grid swaps. This results in a simple and finite action-space. By formulating this re-arrangement task as an MDP (states, actions, rewards and state-transitions), existing RL algorithms can help us in finding optimal sequence of swaps. The advantage of such a learning-based sliding window approach is that the past experiences can be generalized and used to solve new windows. Figure 3.4 summarizes the state, action and reward representations used in this work.

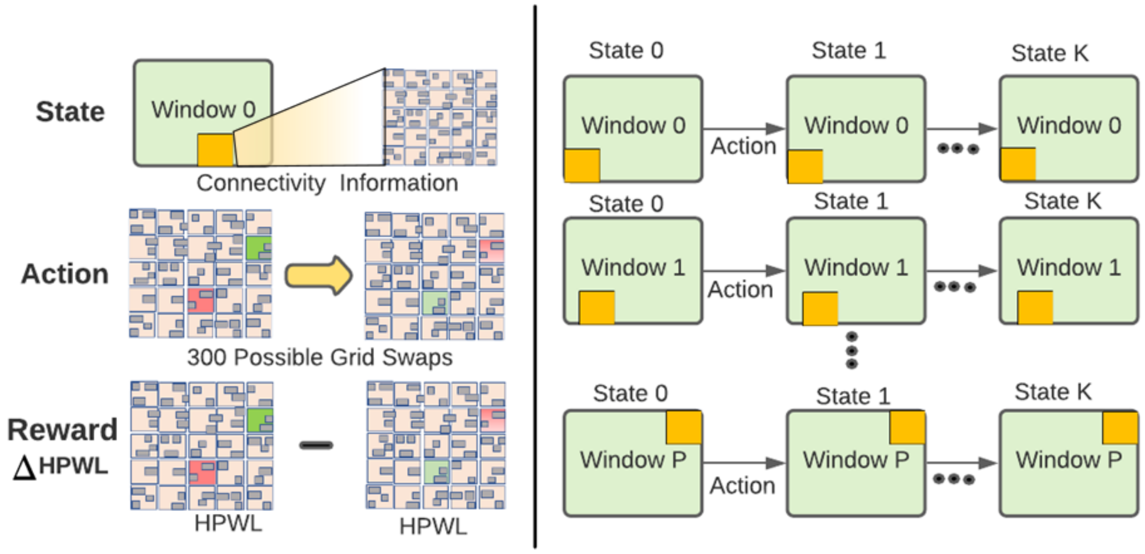


**Figure 3.4.** The components of MDP (left). As the window is slided across the layout, RL agent finds the optimal grid arrangement in each window.

**States**

Since Graph Convolution Networks (GCN) adds a layer of approximation and needs to learn the optimal embedding, we use a simple adjacency matrix of the window for the state representation. The elements of the matrix comprehend the connectivity information between the grids of the window; normalized (for generalizability) summation of pin-pin net connections between the grids. For a window with $k \times k$ grids, the size of adjacency matrix $C$ is $k^2 \times k^2$, where an element $C_{ij}$ indicates the number of pin-pin connections (normalized in 0-1 range) between grid $i$ and grid $j$.

$$s = C_{k^2 \times k^2}, \quad C_{ij} = \frac{\sum Connections(i,j)}{max(C)}$$

**Actions**

We represent the action using $a_{ij} = Swap(i,j)$ indicating the swap of grid $i$ and grid $j$. In a window with $k \times k$ grids, number of actions $|A|$ can be obtained using the following equation.

$$|A| = \binom{k^2}{2}$$

**Rewards**

For minimal wirelength as the objective, one possibility for the reward function is the inverse of wirelength of the associated cells in the window. Since we do not care about absolute goodness while generating the sequence of swaps, we define reward as the HPWL (Half-Perimeter Wire Length) difference of the current state $s$ (in an episode) as compared to the initial state of the episode $s_0$ (from where we initiated the episode). In this milieu, an episode is just a sequence of actions between an initial and terminal state.

$$R(s) = HPWL(s) - HPWL(s_0)$$

### 3.4.2 Stage 2 (fine-grained placement)

During Stage 1, movement of associated cells in these grids could result in cell overlaps. Figure 3.5 show the overlap-free placement problem-instance to the SMT Solver [39]. Formally, given placement site coordinates $(s_x, s_y)$ and grids to be swapped $g1, g2$; with grid origins $(X_{g1}, Y_{g1})$ and $(X_{g2}, Y_{g2})$, grid dimensions $(W_{g1} H_{g1})$ and $(W_{g2}, H_{g2})$, the goal is to determine optimal overlap-free locations $(x_i, y_i)$ for the cells of these two grids. The binary auxiliary variables $l_{ij}, r_{ij}, u_{ij}, d_{ij}$ represent the relative placement between cells $i$,$j$.

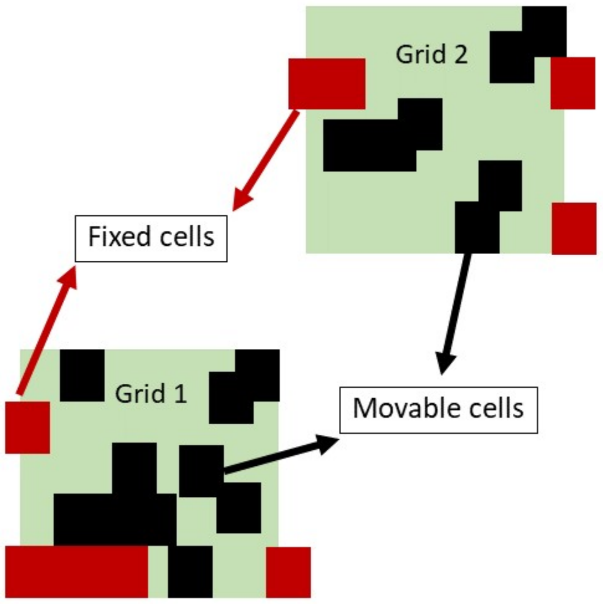


**Figure 3.5.** Stage 2 involves re-arrangement of cells within grids of the window.

**Cell-overlap constraint:**

$$\mathbf{ITE}\left(l_{ij},(x_i+w_i \leq x_j),True\right),\mathbf{ITE}\left(r_{ij},(x_i-w_j \leq x_j),True\right)$$

$$\mathbf{ITE}\left(u_{ij},(y_i-h_j \leq y_j),True\right),\mathbf{ITE}\left(d_{ij},(y_i+h_i \leq y_j),True\right)$$

$$\mathbf{AMO}(l_{ij},r_{ij},u_{ij},d_{ij})$$

**Boundary conditions:**

$$\bigvee\left\{\{(x_i \geq X_{g1})\bigwedge(x_i \leq X_{g1}+W_{g1})\},\right.$$
$$\left.\{(x_i \geq X_{g2})\bigwedge(x_i \leq X_{g2}+W_{g2})\}\right\}$$

**Legal placement (snap to grids):**

$$\left\{(x_i\,mod\,s_x=0)\bigwedge(y_i\,mod\,s_y=0)\right\}$$

**Net connectivity (bounding box):**

For every net $n \in N$, connecting a set of pins $P(n)$, associated with cells $c \in C$ in grids $g1, g2$; the location of a pin $p$ connected by net $n$ is approximated by $(x_p^n, y_p^n) = (x_c^n + 0.5 * w_c, y_c^n + 0.5 * h_c)$, where $(x_c^n,\ y_c^n)$ are the lower-left coordinates of the associated cell $c$, and $(w_c,\ h_c)$ are the cell dimensions. The upper-right and lower-left xy coordinates of a net $n$'s bounding box $ur_x^n$, $ur_y^n$, $ll_x^n$ and $ll_y^n$ can be expressed as:

$$ur_x^n \geq\ x_p^n\ , ur_y^n \geq\ y_p^n\ \forall p \in P(n), n \in N$$

$$ll_x^n \leq\ x_p^n\ , ll_y^n \leq\ y_p^n\ \forall p \in P(n), n \in N$$

**Objective:** Half-perimeter wire length (HPWL)

$$min \sum_{n \in N} \left\{ (ur_x^n - ll_x^n) + (ur_y^n - ll_y^n) \right\}$$

## 3.5 Experimental Setup

For our policy network, we use Deep Neural Network (DNN + Q Learing) to approximate the Q values. The network takes the adjacency matrix of the window as the input and predicts the Q values for each grid-swap action. To train the DNN, we simulate and generate sets of (State, Action, Reward, Next state, Next Action) with an $\varepsilon$-greedy strategy.

**Development Framework:** We use our in-house python-based tool that stores the physical-design database and supports various utilities of a typical RL framework. Since the neural-network libraries and SMT solver is python-based, it offers a single development framework mitigating various tool/data hand-shakes. We perform search over the hyper-parameter space to determine modeling parameters; a fully-connected $300 \times 600 \times 600 \times 300$ neural network, a learning rate of 0.001, discount factor of 0.92, $\varepsilon_{initial}$ of 0.5 and $\varepsilon_{decay}$ of 0.85, replay buffer size of 1000, training batch size of 128, 1000 training episodes and 200 iterations per each

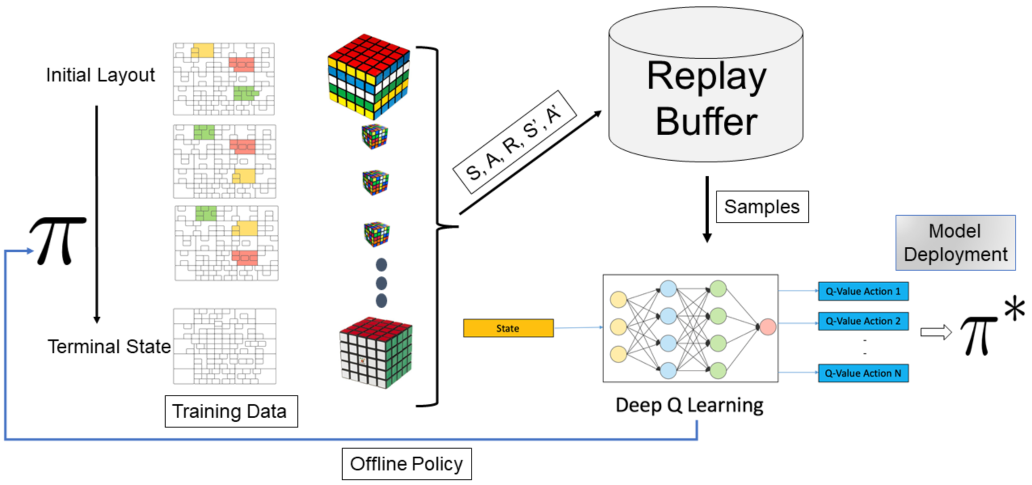


**Figure 3.6.** Optimal re-arrangement problem is analogous to the Rubik's cube problem, with an added complexity of net connectivity between the miniature cubes.

episode.

**Training Setup:** Using the TD update, the policy network is trained to minimize the mean squared loss error (MSE). Each episode of the training process contains a simulated state-transition trajectory, generated with an $\varepsilon$-greedy strategy. The MSE decrease during the training process, the cumulative reward at the end of each training episode and the cumulative reward of the final episode is shown in Figure 3.7. It is important to observe that the model initially explores and eventually learns to exploit good trajectories.

**Inference:** Once the optimal sequence of grid-swaps is determined, we use SMT solver on-the-fly, to assign each cell within the grid to a legal location, without overlaps. To reduce the problem complexity, SMT solver does not disturb the boundary cells since they will be accounted for, as the sliding windows moves over the layout.

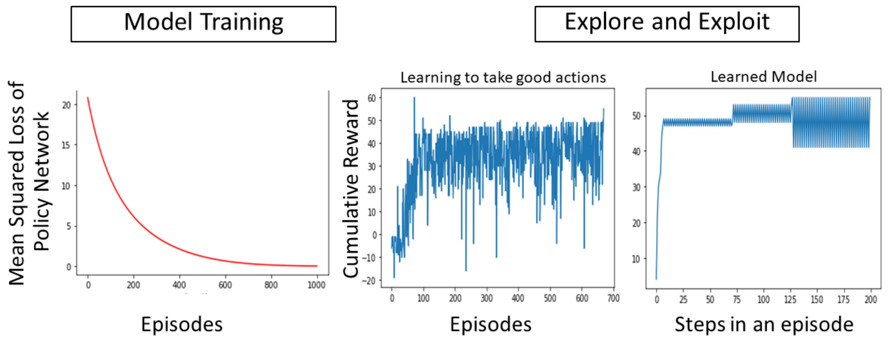


**Figure 3.7.** The MSE of the policy network (left), reward improvement as a function of training episodes (middle) and the final episode's cumulative reward (right).

**Designs:** For our experiments, we use two critical IPs (up to 22K cells) *Multiplier Unit 1* and *Multiplier Unit 2*. In our SOC, we replicate 1000s of these two critical IPs. The improvement in each of these IPs can be scaled by the number of instantiations, to get an estimated improvement on the full-system. We perform two experiments (a) Four optimized global-placement variants of each IP (by varying a combination of physical and logical constraints) serve as the start point. With the same start point, we compare the performance (HPWL) of `RLPlace` with a commercial detailed placer's result. (b) In addition, we also validate if `RLPlace` can improve over two highly-optimized detailed-placement variants resulting from several iterations of tool and manual detailed-placement optimizations.

## 3.6 Experiments

### 3.6.1 Improvement over Global-Placement

As shown in Figure 3.8, we start with a global-placement database from a commercial tool and track the HPWL improvement as the `RLPlace` algorithm is executed. In the plot, X-axis indicates iterations that are ”action sequences” (determined by our RL sequence) in each window along with SMT-based refinement. Once we span the entire layout by window sweeping, we repeat the process of sweeping again (within our runtime budget or till the cost metric saturates). The wirelength results of `RLPlace` as compared to the results of a commercial tool's detailed-

**Table 3.1.** Wirelength (um) comparison of RLPlace with a commercial tool; optimized output from global placement (Baseline) as the start point for both.

| Design | Baseline | Innovus | RLPlace | Delta (%) |
|---|---|---|---|---|
| Multiplier Unit 1 | | | | |
| *design1* | 11900 | 11520 | 11390 | 1.12% |
| *design2* | 12350 | 11900 | 11780 | 1.01% |
| *design3* | 13460 | 12410 | 12280 | 1.04% |
| *design4* | 14250 | 12520 | 12350 | 1.35% |
| Multiplier Unit 2 | | | | |
| *design1* | 133820 | 126290 | 124810 | 1.19% |
| *design2* | 135790 | 128500 | 126940 | 1.23% |
| *design3* | 139530 | 131540 | 130050 | 1.15% |
| *design4* | 142680 | 134610 | 133210 | 1.05% |

placement are summarized in Table 3.1. With the same baseline start point, we observe up to 1.35% HPWL improvement as compared to the results of a commercial tool.

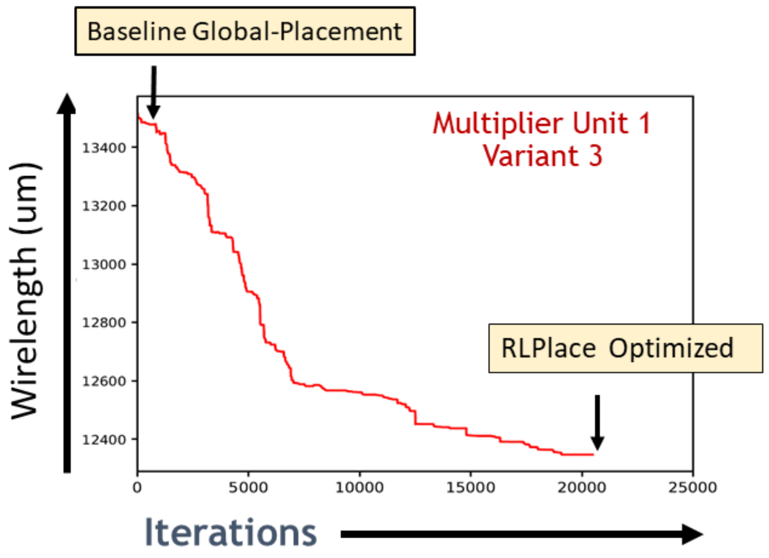


**Figure 3.8.** HPWL improvement of `RLPlace` , with global-placement of *Multiplier Unit 1* as the start point.

.

We generate four variants of both IPs, *design1* and *design2*. These variants are generated by using a combination of design logical constraints (such as utilization, aspect ratio and clock period) while generating the placement data from a commercial tool. We then use the output of commercial tool as the start point for our RLPlace. In our plots shown in Figure 3.8 and Figure 3.9, we observe STWL (in *um*) improvement as a function of iterations (during the first few iterations). When run for sufficiently longer number of iterations, Figure 3.10 shows STWL improvement for a variant of *design1*. This is just a validation of our model's usefulness on unoptimized designs.

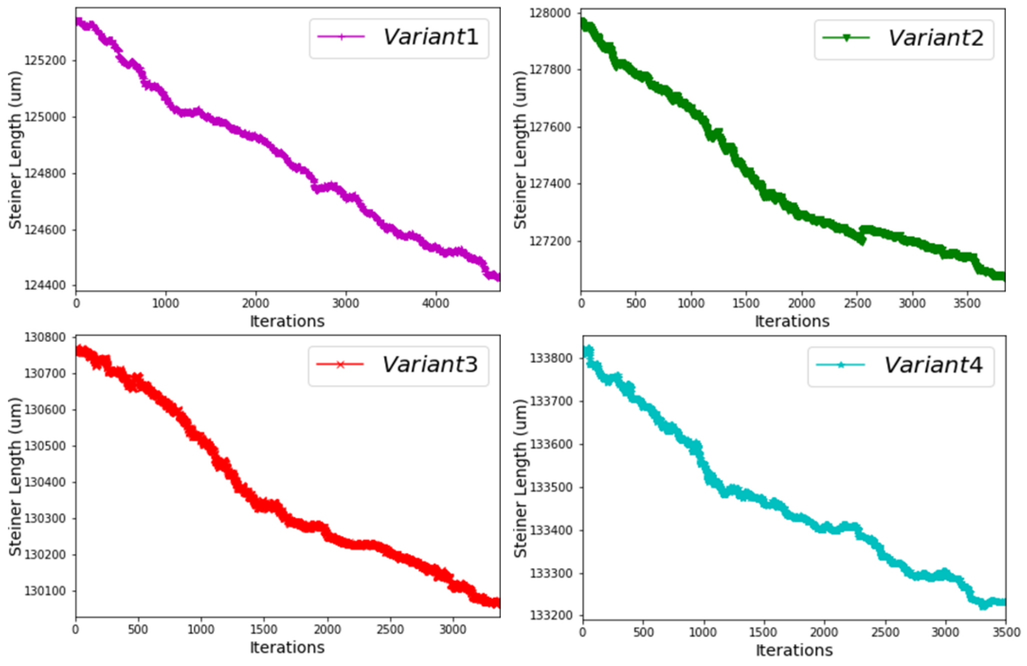


**Figure 3.9.** HPWL improvement during the first 4K iterations (for four variants of *design2*).

.

## 3.6.2 Improvement over Commercial Detailed-Placement Flow

We let the design undergo a commercial tool's detailed-placement optimization, followed by a sequence of manual optimizations. This is used as the start point for our `RLPlace` algorithm. As shown in plots of Figure 3.11, the resulting solution from `RLPlace` shows at least 1.2% HPWL improvement over the heavily optimized design implementations. As shown in Figure 5.10, the end result of `RLPlace` is overlap-free.

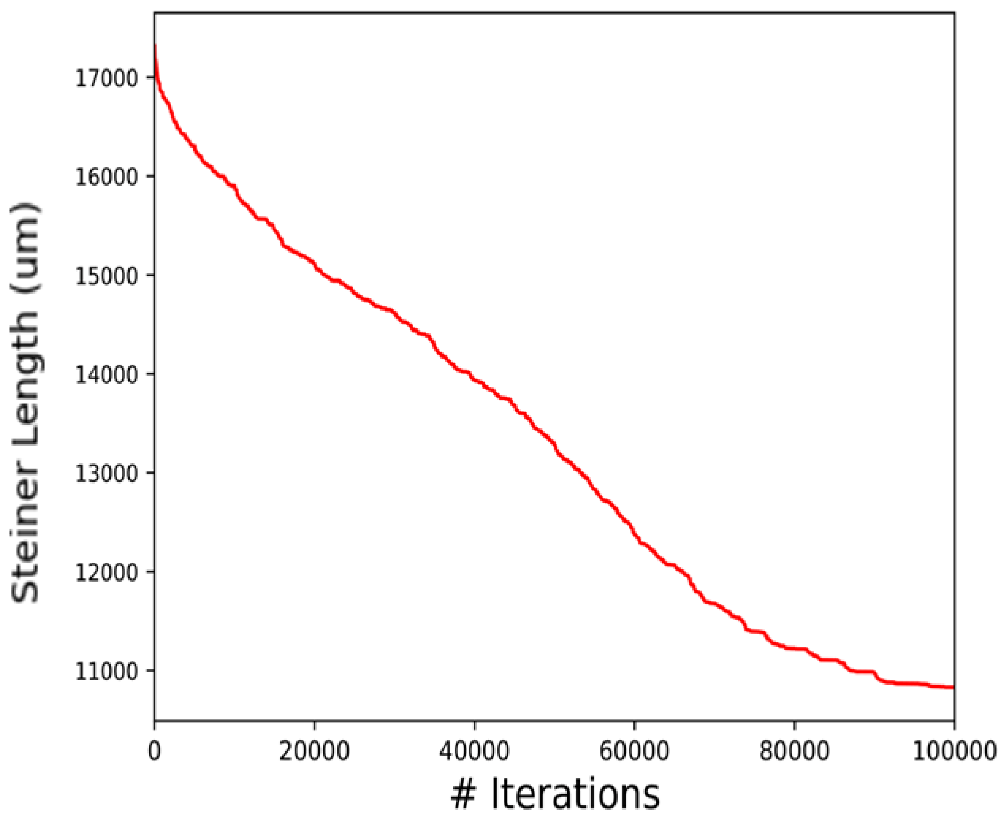


**Figure 3.10.** Plots showing substantial STWL improvement (over 35%) for a variant of *design1*, when run for sufficiently longer number of iterations (we run for 100K iterations for this variant).

.

## 3.7 Conclusion

We use active learning to improve the heuristics of the region-based detailed-placement optimization. With global placement as the start point, we achieve up to 1.35% improvement as compared to commercial tool's detailed-placement. With heavily optimized (commercial tool and manual) detailed-placement as the start point, we achieve˜1.2% HPWL improvement. As part of our future work, we seek to use several interesting state-action representations and simultaneously optimize for multiple rewards (routability, power and performance). We would like to thank Zeki Bozkus, Ghasem Pasandi and James Forsyth of Nvidia Corporation, for their valuable contributions to this work.

Chapter 3 contains materials from "RLPlace: deep RL guided heuristics for detailed

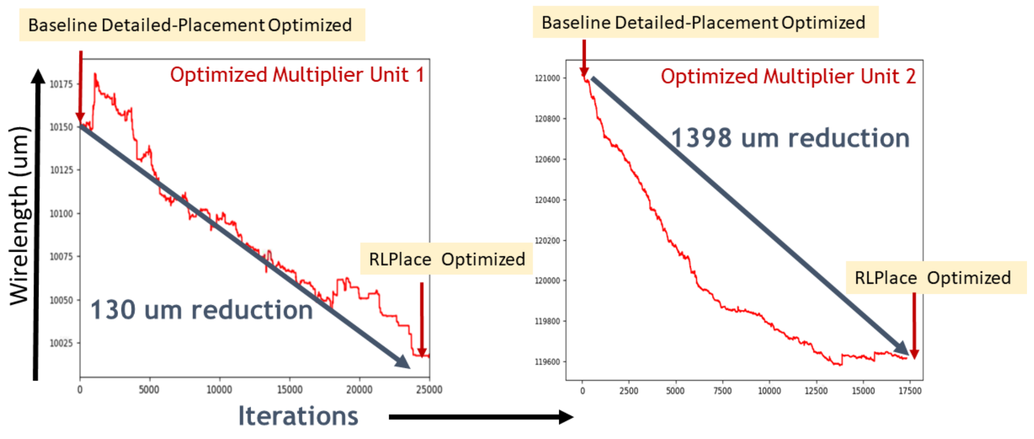


**Figure 3.11.** Plots showing HPWL improvement over highly-optimized design variants of our IPs

.

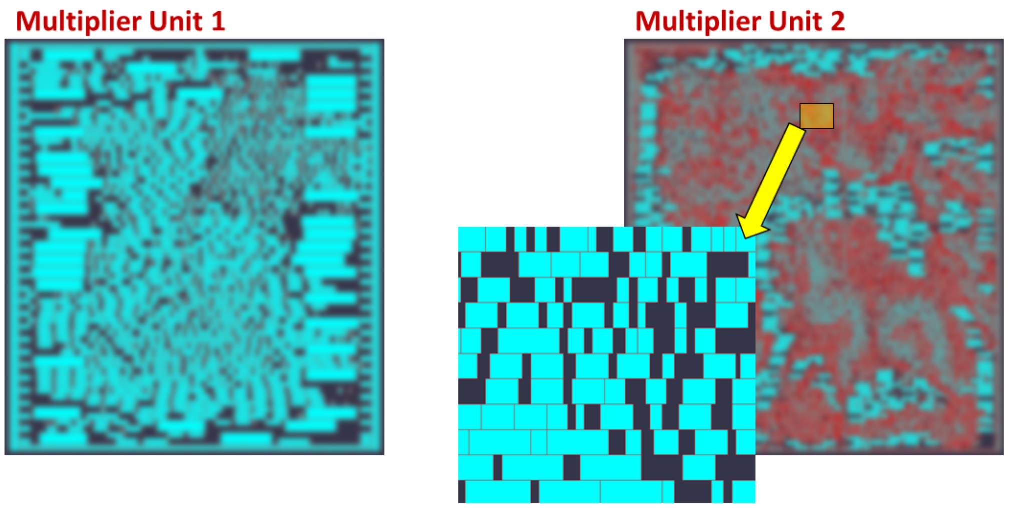


**Figure 3.12.** Overlap-free layout (blurred) of our designs, generated from the RLPlace algorithm.

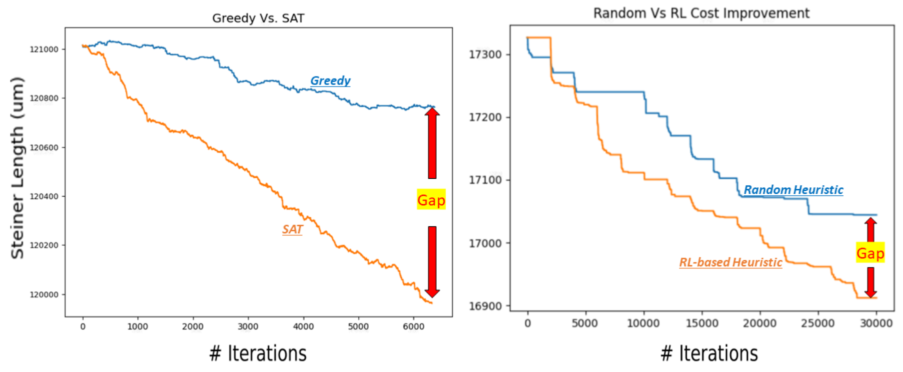


**Figure 3.13.** Plots showing the benefit of SMT/SAT based cell distribution with greedy-swap (Stage 2), and comparison of RL-based heuristic with random heuristics (Stage 1).

placement optimization", by Uday Mallappa, Sreedhar Pratty and David Brown, which appears in Design, Automation and Test in Europe Conference & Exhibition, March 2022. The dissertation author was the primary investigator and author of this paper.

# Chapter 4

# AI for EDA: Network-on-Chip Routing

Application-aware oblivious routing that supports flow-specific routing and static virtual channel assignments can achieve better performance than conventional oblivious routing by accounting for application traffic demands. However, because these prior approaches consider the application-aware routing and static virtual channel assignment steps separately, they unnecessarily impose routing restrictions to avoid deadlocks. We propose a joint application-aware oblivious routing and static virtual channel allocation framework, `JARVA` that optimally solves both problems together to enable better deadlock-free performance. Our approach can achieve up to 30% better performance than the state-of-the-art application-aware oblivious routing algorithms and substantially better still in comparison with conventional oblivious routing.

## 4.1 Introduction

On-chip interconnection networks are widely used in systems-on-chips and multiprocessor designs. Routing algorithms for on-chip interconnection networks can either be oblivious or adaptive. In oblivious routing, the routing path is determined entirely by the source and destination. Oblivious routing is attractive due to its simplicity, which enables simple and fast router implementations. However, conventional oblivious routing can have difficulty with certain communication patterns, especially since routing decisions are made independent of the different flow bandwidth demands for different applications. On the other hand, adaptive routing aims to take into account the network state (e.g., network congestion) when making routing decisions dynamically. Although adaptive routing can in theory achieve better performance if routers can obtain a global and instantaneous view of the network in real-time, it typically relies instead primarily on local information due to speed and complexity limitations, which limits the effectiveness of adaptive routing.

Application-aware oblivious routing [81] is an alternative routing approach to conventional oblivious and adaptive routing. It is particularly well-suited for long-running to sustained applications with predictable traffic patterns. Rather than taking into account the dynamic state of the network, as in the case of adaptive routing, application-aware oblivious routing takes into account the traffic characteristics of an application to pre-compute routes for the different flows so that network performance is optimized, while ensuring deadlock avoidance. The routers in the network are then statically configured with flow-based routing information prior to the execution of the application. In other words, `JARVA` is an "offline" bandwidth-sensitive routing and VC assignment algorithm that optimizes the network performance for a specific application profile, while ensuring deadlock freedom. This approach can achieve better performance than conventional oblivious routing because global knowledge of traffic demands is used to find optimized routing paths. At the same time, routers remain simple because routes are configured statically and remain unchanged at run-time. This approach can achieve better performance than conventional oblivious routing algorithms because routes are optimized with global knowledge of bandwidth demands. At the same time, routers remain simple because routes are configured statically and remain unchanged at run-time.

While pre-computing static flow-specific routes that are optimized for the traffic characteristics of an application can achieve better performance than conventional routing approaches, the chosen routes must also be deadlock free. Previous work on application-aware oblivious routing [81] has considered the application-aware routing and virtual channel (VC) assignment steps separately. To ensure deadlock freedom, Kinsy et al. [81] proposed two approaches. In the first approach called BSOR, Kinsy et al. proposed to restrict the routes in conformance with one of the turn models described in [5]. For a two-dimensional mesh topology, there are 12 possible turn restrictions. BSOR can be enumerated over all 12 possible turn restrictions to pick the best set of routes. In a second approach called BSORM, Kinsy et al. proposed to restrict routes to minimal paths, where deadlock freedom can be ensured by partitioning any arbitrary set of minimal routes into 2 sets of VCs. In both cases, the approaches unnecessarily impose routing restrictions to ensure deadlock avoidance, which significantly limits the design space. Lysne et al. introduced the idea of layered routing [82] to avoid deadlocks: it first generates packet-wise minimal routes and then iteratively performs routing changes while assigning VCs (along the routes) to layers such that the resulting layered dependency graph is acyclic. However, this greedy approach does not provide a global view of the problem.

In this paper, we propose a *joint application-aware oblivious routing and static virtual channel allocation* framework that optimally solves both steps together to enable better deadlock-free performance. By solving both problems jointly, we do not unnecessarily impose routing restrictions – any route can be selected as long as there is a corresponding VC assignment that avoids deadlocks. Our proposed framework is based on a satisfiability modulo theories (SMT) formulation that enables the succinct capture of complex conditional constraints that are needed to model our joint optimization problem, especially the complex interactions between demand-sensitive routing and deadlock free VC assignment. Our SMT framework can support various optimization cost functions, including maximizing the satisfaction of flow demands or minimizing the maximum channel load, while ensuring deadlock avoidance. We evaluate our framework on both synthetic benchmarks and a real-world case study. The experimental results show that our approach can achieve up to 30% better performance than the state-of-the-art application-aware oblivious routing algorithms and substantially better still in comparison with conventional oblivious routing approaches.

The remainder of this paper is organized as follows: Section 4.2 summarizes related work. Section 4.3 describes our SMT-based formulation for our joint application-aware oblivious routing and static virtual channel allocation problem. Section 4.4 compares benchmark performance using our joint routing and VC allocation approach to existing application-aware and deterministic oblivious routing algorithms. Finally, Section 4.5 concludes the paper.

## 4.2 Related Work

Dimension-order routing [83] is one of the deterministic routing methods that manifests into XY or YX routing in a 2D network grid. Necessary and sufficient conditions for a deadlock-free deterministic routing were given in [84] in scenarios where there are no false resource dependencies. We use this condition to determine if a set of routes is deadlock-free in our oblivious routing scheme. In addition to XY or YX routing, ROMM [85] and Valiant [86] use randomized intermediate node selection, to improve the load distribution. In O1TURN routing algorithm [87], Seo *et al* showed that simply balancing traffic between XY and YX routing can guarantee near-optimal worst-case throughput. These application independent oblivious routing algorithms are indifferent to the traffic pattern. On the other hand, application-aware oblivious routing [81] is concerned with optimizing throughput for specific (possibly long-running) applications by considering application-specific traffic patterns and bandwidth requirements.

Given an application, existing methods to obtain deadlock free routes can be divided into heuristic and exact methods. For the specific problem of application-specific oblivious routing, Kinsy *et al* [81] propose to first solve the routing task using MILP. However, due to its scalability issues, they propose BSOR (single virtual channel or multiple virtual channels with dynamic VC allocation) and BSORM (multiple virtual channels with minimal routes and therefore making static virtual channel allocation) to determine the lower bound of edge residual capacity. The routing scheme in BSOR and BSORM relies on the order of flow routes chosen during their sequential flow routing process and makes it a tedious process, when the gap between upper (usually chosen from XY or YX routing results) and the lower bound of residual edge capacity is very large. In addition, when virtual channels $\geq 2$, BSORM-based routing lifts flow-turn restrictions, but at the cost of restricting to minimal routes (to ensure deadlock-freedom). Unlike previous works for oblivious routing which either solve for exact solutions in networks with smaller complexities or use heuristics at the cost of optimality, we propose SMT-based exact solver to solve for deadlock-free routing and VC assignment simultaneously.

## 4.3 Joint Application-Aware Oblivious Routing and Static VC Assignment (`JARVA`)

In this section, we first summarize the notations for a 2D mesh network model. We then outline how deadlocks can be avoided in a 2D mesh in the case of wormhole routing (the case with no virtual channels) based on a turn restriction model [5]. We next extend the conditions to sufficiently guarantee deadlock freedom in the case of static VC assignment. Based on these conditions, we provide an SMT formulation of our *joint* application-aware oblivious routing and static VC assignment problem, which captures complex conditional constraints that are needed to model our joint optimization problem, especially the complex interactions between demand-sensitive routing and deadlock free VC assignment. We conclude this section by discussing extensions to the router implementation to support application-aware oblivious routing and static VC assignment.

### 4.3.1 Network Model

An $M \times N$ 2D mesh network can be represented by a directed graph $G = (V, E)$, where each node $u \in V$ corresponds to a processing core and an associated router. Each node has a location coordinate $(m, n)$, where $m$ is the row number and $n$ is the column number with the origin $(0, 0)$ at lower-left node of the network. In general, a node $u$ at location $(m, n)$ has a directed link $(u, v) \in E$ to an adjacent node $v$ at location $(p, q)$ as long as $0 \leq p \leq (M-1)$ and $0 \leq q \leq (N-1)$.

For the application-aware oblivious routing problem and static VC assignment problem considered in this paper, application requirements are given as a set of $K$ flows, indexed by

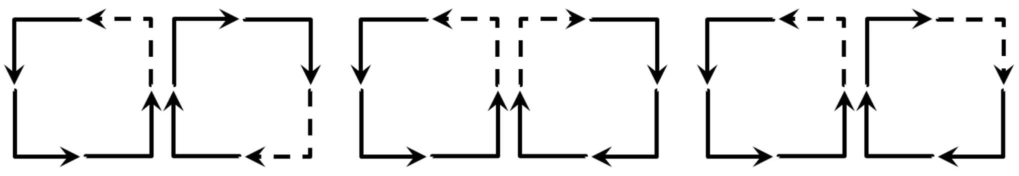


**Figure 4.1.** The turns (clockwise and counter-clockwise) that are allowed (solid lines) and prohibited (dashed lines) under (a) the West-First turn model, (b) the North-Last turn model, and (c) the Negative-First turn model, each with four rotations (steps of 90 degrees), for a total of 12 different turn models [5].

$i = 1,2,\ldots,K$. Each flow $i$ is defined by $(s_i,t_i,d_i)$, where $s_i$ and $t_i$ are the source and destination, respectively, and $d_i$ is the demand. We assume $s_i \neq t_i$ and *multiple flows may have the same source and destination*. A route for a flow $i$ is a path from $s_i$ to $t_i$.

### 4.3.2 Deadlock Avoidance in a 2D Mesh in Wormhole Routing

For deadlock avoidance, [5] showed that deadlocks can be avoided in wormhole routing (the case with no virtual channels) by restricting the turns that a flow can take in its routing path. Fig. 4.1 shows three types of turn models that can ensure deadlock avoidance in wormhole routing. Fig. 4.1a illustrates one turn model called West-First in which all turns in the West direction are prohibited in either the counter-clockwise or clockwise direction. The other two types of turn models called North-Last and Negative-First are similarly depicted in Fig. 4.1b and Fig. 4.1c, respectively. These turn models are explained in depth in [5]. As each turn model type can be rotated four different ways, there are in effect 12 different turn models.

**Proposition 1** *For any of the 12 possible turn restriction models, any set of routes that obey the chosen turn restriction model is deadlock free for wormhole routing.*

The proofs were given in [5] for each of the West-First, North-Last, and Negative-First turn model, which also apply to the corresponding four possible rotations (steps of 90 degrees) of each model, for a total of 12 possible turn restriction models. The proofs were derived from a key result from Dally and Seitz [84] that showed any routing algorithm is deadlock free if the channels in the network can be numbered so that the algorithm routes every packet along channels with strictly increasing (or decreasing) numbers. In [5], Glass and Ni showed how the channels in a 2D-mesh can be numbered for each of the turn models so that the channels would be strictly increasing (or decreasing) as long as the consecutive links obey the corresponding turn restrictions. In particular, in [5], Glass and Ni assumed wormhole routing, where each channel has a one-to-one correspondence to a link. In this case, the numbering scheme is such that the corresponding channels on two adjacent links are numbered in strictly increasing (or decreasing) order, except for the prohibited turns, which would have decreasing (or increasing) numberings. We refer the interested reader to [84] and [5] for details[1]. Note that [5] assumes wormhole routing (the case with no virtual channels). Therefore, each channel has a one-to-one correspondence with a link in the 2D mesh.

The application-aware oblivious routing approach in [81] called BSOR adopts this turn model approach by restricting the routing of flows in conformance with one of these 12 turn models. BSOR can be enumerated over all 12 possible turn restrictions to pick the best set of routes. BSOR imposes routing restrictions on flows because it performs the application-aware oblivious routing step and the static VC assignment step *separately*. Therefore, it has to perform the routing assignment of flows as if deadlocks could not be resolved by virtual channel assignments. In our work, we *do not impose turn restrictions on flows*. Rather, the key idea in our approach is that we allow flows to make such restricted turns as long as the VCs statically assigned to the flow on the two links corresponding to the restricted turn (dotted lines in Figure 4.1) are strictly *increasing* with respect to the *indices* assigned to the VCs (see the next section).

[1]In [5], the channels were numbered in decreasing order and the turn restrictions ensured that consecutive links (channels) on a route were strictly decreasing, but exactly the same analysis applies with the channels numbered in increasing order and consecutive links (channels) were required to be in strictly increasing order.

### 4.3.3 Deadlock Avoidance in a 2D Mesh with Static VC Assignment

In this section, we extend the turn model restrictions in [5] for the case where each link has $H$ virtual channels, for any $H \geq 2$. We first define the following.

**Definition 1** *A channel dependency graph (CDG) is a directed graph $D = (C, A)$, where the vertex set $C$ corresponds to a set of channels in a network and the arc set $A$ consists of pairs of channels $(c_i, c_j)$, $c_i, c_j \in C$, such that there is a direct dependency from $c_i$ to $c_j$ if there is routing of a flow that transitions from channel $c_i$ to $c_j$.*

**Definition 2** *A layered channel dependency graph (L-CDG) for a 2D mesh $G = (V, E)$ with $H$ virtual channels per link is a channel dependency graph $D = (C, A)$, with the following properties:*

1. *Each link $e \in E$ in the 2D mesh $G = (V, E)$ has exactly $H$ virtual channels, indexed $h = 1, 2, \ldots, H$.*

2. *The vertex set $C$ in the L-CDG has exactly $|E| \times H$ channels in the network, where the channels are disjointly partitioned into $H$ layers, $L_1, L_2, \ldots, L_H$, and where each layer $L_h = \{c_1^h, c_2^h, \ldots, c_{|E|}^h\}$ comprises the virtual channels with index $h$ for all $|E|$ links in $G = (V, E)$ such that $c_e^h$ is a virtual channel on link $e$ with VC index $h$.*

3. *There is a direct dependency in the arc set $A$ from $c_i^g$ to $c_j^h$ if there is a routing of a flow that transitions along two consecutive links from channel $c_i^g$ to $c_j^h$, where $g$ and $h$ could be on the same layer (i.e., $g = h$).*

The key idea in our approach is to extend the turn restriction framework in [5] to allow flows to make restricted turns as long as the VCs statically assigned to the flow on the two links corresponding to the restricted turn (dotted lines in Figure 4.1) are always strictly *increasing* with respect to the *indices* assigned to the VCs. That is, we perform application-aware oblivious routing and static VC assignment *jointly* to ensure the following conditions are satisfied[2]:

**Condition 1** *If a route on two consecutive links correspond to a restricted turn, the VC index assigned to the second link must be strictly increasing.*

**Condition 2** *If a route on two consecutive links does not correspond to a restricted turn, the VC index assigned to the second link can remain the same or be strictly increasing.*

**Proposition 2** *If a joint application-aware oblivious routing and static VC assignment for a set of flows satisfies the above conditions for a 2D mesh with H virtual channels per link, then the corresponding solution is deadlock free.*

As defined in Definition 2, the channels in an L-CDG $D = (C, A)$ are organized into $H$ layers, $L_1, L_2, \ldots, L_H$. Conceptually, we can view an L-CDG as being *three-dimensional*, with all virtual channels having the same index $h$ grouped on to the same layer $L_h$. As stated in Conditions 1 and 2, *when a VC assignment is changed* on two consecutive links of a route, the VC index assigned to the second link must be strictly increasing. A change in the VC assignment is only required on a restricted turn; otherwise, it is optional .

Intuitively, deadlocks are avoided on a VC layer by requiring the VC assignment to go up to a higher indexed VC layer on restricted turns, thereby making it impossible to have cyclic dependencies on a layer. That is, it has already been shown in [5] that cyclic dependencies cannot occur if restricted turns are prohibited. In our L-CDG case, cyclic dependencies cannot occur among the channels on the same layer since we require the VC assigned to the second link on a restricted turn to transition to a strictly higher VC index (see Condition 1).

[2]In these conditions, the VCs are assumed to be indexed as $h = 1, 2, \ldots, H$: *strictly increasing* means for example, if the first link is assigned $h = 1$, then the second link must be assigned $h \geq 2$. The conditions could also be defined so that when the VC index of the second link is changed, the new VC index must be *strictly decreasing*. This is equivalent to *strictly increasing* if we simply reverse the numbering of the VCs. Similarly, the VCs can be renumbered under any permutation, and the notion of *strictly increasing* applies accordingly. Therefore, for the remainder of the paper, we will simply assume the new VC index must be strictly increasing when the VC index of the second link is changed, but any permutation of the ordering would also ensure deadlock freedom under the same conditions.

**Table 4.1.** Terms and Definitions.

| Term | Definition |
|---|---|
| $\alpha_i(u,v)$ | A Boolean (0/1) variable indicating if flow $i$ is assigned to link $(u,v)$ |
| $\beta_i^c(u,v)$ | A Boolean (0/1) variable indicating if flow $i$ is assigned to link $(u,v)$ using VC with index $c$ |
| $load(u,v)$ | Total load on link $(u,v)$ |
| $MCL$ | The maximum channel load |

Next, we have to consider if cyclic dependencies can occur among the channels on different layers. Both Conditions 1 and 2 allow for link transitions to a strictly higher VC index. Because the VC assignments for consecutive links must always go up to a higher VC layer when changed, deadlocks also cannot occur *between* layers since the VC assignments for consecutive links can never go back down to a lower layer, thereby making it impossible to have cyclic dependencies between layers.

Therefore, the resulting L-CDG is guaranteed to be acyclic, which guarantees the corresponding routing and static VC assignment is also deadlock free.

Based on the above conditions, we provide in the next section a novel SMT formulation that jointly performs application-aware oblivious routing and static virtual channel allocation as a global optimization problem. The formulation considers all flows simultaneously to enable better deadlock-free performance and allows us to capture complex conditional constraints that are needed to model our joint optimization problem, especially the complex interactions between demand-sensitive routing and deadlock free VC assignment.

### 4.3.4 SMT Formulation

SMT is well-suited to formulate our joint optimization problem as constraints and optimization criteria involving Boolean variables, integer variables, and real variables can be readily expressed. As described below, an At-Most-One" (**AMO**) constraint over a set of Boolean variables imposes the requirement that at most one of the Boolean variables can be true (set to 1), and an "Exactly-One" (**EO**) constraint over a set of Boolean variables imposes the requirement that exactly one of the Boolean variables is true (set to 1). Another useful type of logical constraints is the "If-Then-Else" ($\mathbf{ITE}(a,b,c)$) constraint, which returns $b$ if $a$ is true and $c$ otherwise. Logical operations like conjunctions, disjunctions, or implications can be readily specified as well. Table 4.1 summarizes the notation that we use in our formulation, and the SMT formulae are given below.

**Flow conservation**: For source and destination nodes, each flow $i$ should be assigned to *exactly one* outgoing link $(s_i, v)$ emanating from source $s_i$ and *exactly one* incoming link $(w, t_i)$ going into destination $t_i$.

$$\forall i, \quad \underset{(s_i,v)\in E}{\mathbf{EO}} \left\{ \alpha_i(s_i, v) \right\} \forall i, \quad \underset{(w,t_i)\in E}{\mathbf{EO}} \left\{ \alpha_i(w, t_i) \right\} \tag{4.1}$$

For a non-source or a non-destination node $u$, each flow $i$ should be assigned to *at most one* link among all links $(v, u)$ going into $u$, and each flow $i$ should be assigned to *at most one* link among all links $(u, w)$ emanating from $u$. If flow $i$ is assigned to a link going into $u$, then flow $i$ should also be assigned to a corresponding outgoing link from $u$, and vice versa.

$$\forall i, \forall u \neq s_i, t_i, \quad \underset{(v,u)\in E}{\mathbf{AMO}} \left\{ \alpha_i(v, u) \right\} \ \& \ \underset{(u,w)\in E}{\mathbf{AMO}} \left\{ \alpha_i(u, w) \right\} \tag{4.2}$$

$$\forall u \neq s_i, t_i, \quad \bigvee_{(v,u)\in E} \alpha_i(v, u) = \bigvee_{(u,w)\in E} \alpha_i(u, w) \tag{4.3}$$

The per-flow constraints in Equations 4.1-4.3 together ensure that every flow is connected from its source to destination *without loops* and routed along a single path as an unsplittable flow and that every route is possible, including non-minimal paths.

**Static VC assignment**: If flow $i$ is assigned to a link $(u, v)$, then it must also be statically assigned to *exactly one* of its VCs.

$$\forall (u, v) \in E, \quad \alpha_i(u, v) \Rightarrow \underset{\forall c}{\mathbf{EO}} \left\{ \beta_i^c(u, v) \right\} \tag{4.4}$$

$$\forall (u,v) \in E, \quad \neg \alpha_i(u,v) \Rightarrow \neg \bigvee_{\forall c} \left\{ \beta_i^c(u,v) \right\} \tag{4.5}$$

**VC constraints to ensure deadlock avoidance**: As explained in Section 4.3.2 and Fig. 4.1, there are three different turn models, West-First, North-Last, and Negative-First, each with four rotations, for a total of 12 different turn models [5]. Any of these 12 turn models can be used to ensure deadlock avoidance. $T$ denotes the *chosen turn model* for the entire system, and each pair of links $\{(v,u),(u,w)\} \in T$ denotes a *turn* that is *disallowed* under that turn model. However, unlike [81], we *do not impose turn restrictions on flows*, which limits the search space of possible routing of flows. Rather, the key idea in our approach is that we allow flows to make such turns as long as the VC assigned to the flow transitions from a lower-indexed VC to a *higher*-indexed VC (*increasing*) at the restricted turn (Equation 4.6). Also, at every non-sink and non-source node, the VC transition is always *non-decreasing* (Equation 4.7), meaning that the VCs assigned on two consecutive links for a flow can remain the *same* or change to a *higher*-indexed VC. This way, we do not impose *any* restriction on the routes that a flow can take, but instead, we appropriately perform simultaneous VC assignment during the routing process. This key idea enables our approach to achieve significantly better results.

$$\begin{aligned} &\forall i, \forall \{(v,u),(u,w)\} \in T, \\ &\bigwedge_{\forall (c_1,c_2),\, c_2 \leq c_1} \mathbf{AMO} \left\{ \beta_i^{c_1}(v,u), \beta_i^{c_2}(u,w) \right\} \end{aligned} \tag{4.6}$$

$$\begin{aligned} &\forall i, \forall u \neq s_i, t_i \\ &\bigwedge_{\forall (c_1,c_2),\, c_2 < c_1} \mathbf{AMO} \Big\{ \bigvee_{(v,u)\in E} \beta_i^{c_1}(v,u), \\ &\qquad\qquad\qquad\qquad \bigvee_{(u,w)\in E} \beta_i^{c_2}(u,w) \Big\} \end{aligned} \tag{4.7}$$

**Load on each link**: The load on each link is the sum total of the demands of flows assigned to it –

i.e., if flow $i$ is assigned to link $(u,v)$, then the corresponding demand $d_i$ gets added to $load(u,v)$.

$$\forall(u,v) \in E, \quad load(u,v) = \sum_{i=1}^{K} \mathbf{ITE}\Big(\alpha_i(u,v), d_i, 0\Big) \tag{4.8}$$

**Maximum channel load**: The maximum channel load (MCL) corresponds to the highest $load(u,v)$ among all links.

$$\forall(u,v) \in E, \quad MCL \geq load(u,v) \tag{4.9}$$

$$\min MCL \tag{4.10}$$

Like [81], we focus in this work on finding the minimum maximum channel load because, as shown in [81], minimizing the MCL correlates well with optimizing for network throughput and network latency. In particular, the injection rate is inversely proportional to MCL with respect to the link capacity: i.e., the injection rate as a multiple of flow demands is $\lambda =$ link capacity$/MCL$.

The above SMT formulation can be enumerated over all 12 possible turn models at the VC level to pick the best set of routes. It should be noted SMT [39] provides a much more expressive modeling language than would be possible with ILP or MILP formulas. Built-in Boolean cardinality functions such as **EO** and **AMO** make it possible to succinctly capture constraints like Equations 4.1, 4.2 and 4.4, and logical implications and the built-in **ITE** function make it possible to succinctly capture constraints like Equation 4.8. These expressive modeling capabilities are essential in modeling our joint routing and VC assignment problem. Further, because modern SMT solvers like Z3 [39] are built on top of a Boolean satisfiability (SAT) solver, they are particularly good at navigating through search problems that are dominated by Boolean variables.

**Alternative Objectives:** In addition to the MCL objective of Equation 4.10, we consider several alternative objectives here. In particular, SMT solvers [39] support lexicographic priority of objectives, where the solver first optimizes for the objective that is declared first, and then incrementally (without degrading the prior objectives) optimizes the objectives that are declared later. For example, Equation 4.11 favors minimal paths or minimal network-resource paths. Furthermore, we can prioritize minimal routes for flows with larger demand using Equation 4.12 or favor routes with larger source-destination distance using Equation 4.13. In Equation 4.13, $distance_i$ refers to the Manhattan distance between the source-destination pair $(s_i, t_i)$ of flow $i$.

$$\min \sum_{i=1}^{K} \sum_{(u,v) \in E} \alpha_i(u,v) \tag{4.11}$$

$$\min \sum_{i=1}^{K} \sum_{(u,v) \in E} \alpha_i(u,v) * d_i \tag{4.12}$$

$$\min \sum_{i=1}^{K} \sum_{(u,v) \in E} \alpha_i(u,v) * distance_i \tag{4.13}$$

**Formulation Complexity:** Table 4.2 presents the formulation complexity of JARVA using the SMT framework for a 2D Mesh. The number of constraints and literals are related to the number of nodes in the network $|V|$, the number of edges in the network $|E|$, the number of flows $|K|$ and the number of virtual channels $|C|$ in the network. Recall that the concise representation in SMT gets rid of auxiliary variables and therefore reducing the formulation complexity. As can be seen in Section 4.4, our formulation can successfully solve a number of widely used benchmarks and a real-life camcorder application on a practical 2D-mesh configuration.

**Table 4.2.** JARVA 's formulation complexity.

| **Constraint Definition** | **# Constraints** | **# Literals per constraint** |
|---|---|---|
| Flow conservation | $2 * \|V\| * \|K\|$ | 8 |
| Static VC assignment | $2 * \|E\| * \|K\|$ | $\|C\|$ |
| Deadlock-freedom | $\|V\| * \|K\|$ | $\|C\|^2$ |
| Load constraint | $\|E\|$ | $\|K\|$ |
| Load bound | $\|E\|$ | 1 |

### 4.3.5 Router Implementation

To support arbitrary routes for different flows and static VC assignment, typical VC-based router microarchitectures [88] can be extended. In particular, previous work [81] showed that typical VC-based routers [88] can be readily extended to support arbitrary per-flow oblivious routing by using a programmable table-based routing mechanism where pre-computed routes between pairs of nodes are stored in a routing table. Further, static VC assignments can be readily implemented by specifying the VC assignment in the same routing table. This static VC assignment approach can improve router performance as the latency of a pipelined virtual-channel router is often dominated by VC allocation when performed dynamically [88]. We refer the reader to [81] for router extensions to support arbitrary routing and static VC assignments. As mentioned above, the key difference with our approach is that we perform application-aware oblivious routing jointly with static VC assignment, which eliminates unnecessary restrictions on the search space of routing choices, leading to better results.

## 4.4 Evaluation

### 4.4.1 Experimental Setup

We evaluate the performance of our joint optimization approach JARVA against several conventional oblivious routing algorithms, including XY and YX dimension ordered routing [83], ROMM [85], Valiant (VAL) [86], and O1TURN [87]. We evaluate both XY and YX separately as the results are different for asymmetric traffic patterns. We also compare with the state-of-the-art application-aware oblivious routing algorithms BSOR and BSORM [81]. As ROMM, VAL, O1TURN, BSORM all require 2 VCs to ensure deadlock avoidance, we also set the number of VCs to 2 VCs for JARVA and for XY or YX routing. For BSOR, we enumerate it over all 12 possible turn models shown in Fig. 4.1 at the flow level and report the best results. We also enumerate JARVA over all 12 possible turn models, but we *do not impose turn restrictions on flows*. As explained in Section 4.3, JARVA avoids deadlocks via constraints on the joint static

**Table 4.3.** Synthetic Traffic Patterns

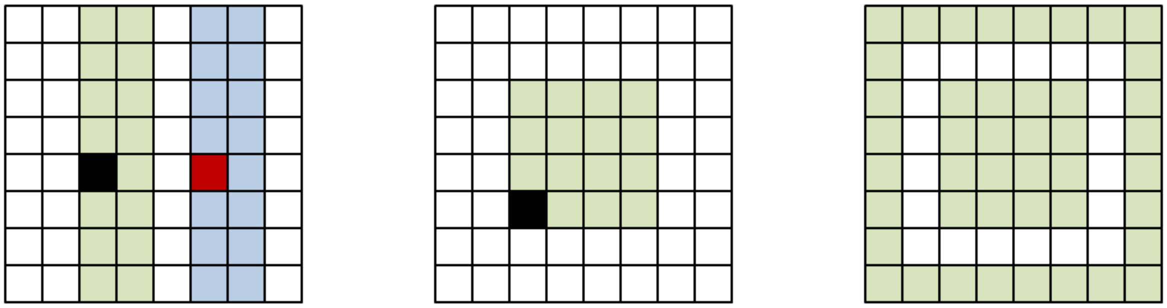
| | |
|---|---|
| Transpose | Packets at $(m,n)$ sent to $(n,m)$ |
| Bitcomp | Packets at $(m,n)$ sent to $(M-m-1,N-n-1)$ |
| Shuffle | Left rotate entire bit-vector representation. e.g., packets at $(011,100)$ sent to $(111,000)$. |
| Hotspot1 | Packets at a black (red) node sent to green (blue) node chosen at uniform random. |
| Hotspot2 | Packets at a black node sent to a green node chosen at uniform random. |
| P-Transpose | Packets at a green node $(m,n)$ sent to $(n,m)$. |

VC assignments. For our evaluations, we use both synthetic benchmarks as well as a real-life camcorder application [89], as detailed below.

### 4.4.2 Synthetic Traffic

We first evaluate the different algorithms using the synthetic traffic patterns shown in Table 4.3, including Transpose, Bitcomp (bit-complement), and Shuffle. In addition, we use the three synthetic patterns depicted in Fig. 4.4. The first two are hotspot traffic patterns, whereas the third one is a variant of the Transpose pattern in which some nodes (in white) do not send traffic. To provide a more realistic setting, we vary packet sizes between 1-6 flits per packet, chosen at uniform random, and we perform the evaluation on an 8×8 mesh. The results are shown in Table 4.5. In all cases, `JARVA` achieves the best results (*), by as much as 30% better than the state-of-the-art application-aware oblivious routing algorithms and substantially better still in comparison with conventional oblivious routing algorithms.

**Interpretation of Results:** Table 4.5 presents the maximum channel load of the network for various combinations of routing algorithms and traffic patterns. We start with a hypothesis that the routing algorithm with lower maximum channel load can withstand higher injection rates before reaching network saturation. Quantitatively, with a single VC, a routing algorithm with a maximum channel load of “6” can withstand a traffic injection rate of $1/6$ as compared to routing algorithm with a maximum load of “7” (corresponding to traffic injection rate of $1/7$). Our hypothesis is validated with flit-level simulation results discussed in Section 4.4.



**(a)** Hotspot1. **(b)** Hotspot2. **(c)** P-Transpose.

**Figure 4.2.** Synthetic traffic patterns. See Table 4.3 for descriptions.

**Table 4.4.** Comparison of Maximum Channel Load on Synthetic Benchmarks.

| Traffic | XY | YX | ROMM | VAL | O1TURN | BSOR | BSORM | JARVA | Δ |
|---|---|---|---|---|---|---|---|---|---|
| Transpose | 17 | 16 | 14 | 17 | 13 | 8 | 7 | 6* | 14.28% |
| Bitcomp | 10 | 10 | 19 | 16 | 14 | 11 | 12 | 8* | 20.00% |
| Shuffle | 16 | 15 | 17 | 27 | 16 | 11 | 10 | 7* | 30.00% |
| Hotspot1 | 29 | 30 | 26 | 53 | 23 | 17 | 20 | 15* | 11.76% |
| Hotspot2 | 36 | 43 | 30 | 35 | 27 | 15 | 21 | 13* | 13.33% |
| P-Transpose | 19 | 17 | 16 | 16 | 13 | 7 | 8 | 6* | 14.28% |

**Table 4.5.** MCL Comparison on Synthetic Benchmarks.

| Traffic | TP | BC | SF | H1 | H2 | PT |
|---|---|---|---|---|---|---|
| XY | 17 | 10 | 16 | 29 | 36 | 19 |
| YX | 16 | 10 | 15 | 30 | 43 | 17 |
| ROMM | 14 | 19 | 17 | 26 | 30 | 16 |
| VAL | 17 | 16 | 27 | 53 | 35 | 16 |
| O1TURN | 13 | 14 | 16 | 23 | 27 | 13 |
| BSOR | 8 | 11 | 11 | 17 | 15 | 7 |
| BSORM | 7 | 12 | 10 | 20 | 21 | 8 |
| JARVA | 6* | 8* | 7* | 15* | 13* | 6* |
| Δ | 14.28% | 20% | 30% | 11.76% | 13.33% | 14.28% |

### 4.4.3 Camcorder Scenarios

We next consider a real-life camcorder application studied in [89], where they considered a 2D mesh network architecture with heterogeneous cores for a mobile processor design. In addition to a CPU core, the heterogeneous processor architecture integrates multiple multimedia cores (shown in orange in Fig. 4.5), such as GPU, DSP, video codecs, JPEG encoders, and camera modules, and multiple system cores (shown in blue in Fig. 4.5), such as a USB controller, a WiFi modem, and a GPS module. The network traffic is dominated by memory traffic to external memory via the memory controller (MC). The bandwidth requirements vary from 33 MB/s to 3.93 GB/s (flow demands are scaled such that 33 MB corresponds to 1 unit of flow). We refer the reader to [89] for a detailed discussion regarding the characteristics of the camcorder application, the heterogeneous cores, and the memory traffic model. Following [89], we also consider four different $5 \times 5$ network configurations, as depicted in Fig. 4.5. The first three configurations, as depicted in Fig. 4.5a-Fig. 4.5c, correspond to different symmetric mesh configurations. The last one depicted in Fig. 4.5d corresponds to an asymmetric configuration in which some routers are used to clustered together multiple cores based on their functions. As shown in Table 4.6, JARVA achieves the best results (*), with 8-21.29% improvements over the state-of-the-art application-aware oblivious routing algorithms and substantially better still in comparison with conventional oblivious routing algorithms.

**Interpretation of Results:** Table 4.6 shows the reduced maximum channel load values of JARVA (for example, 207 vs BSORM's 263), indicating that the network with JARVA 's solution can withstand a higher injection rate before reaching network saturation. Intuitively, $\Delta$ improvements indicate the magnitude of tolerable minimum injection rate increase as compared with previous works.

### 4.4.4 Detailed Flit-Level Simulations

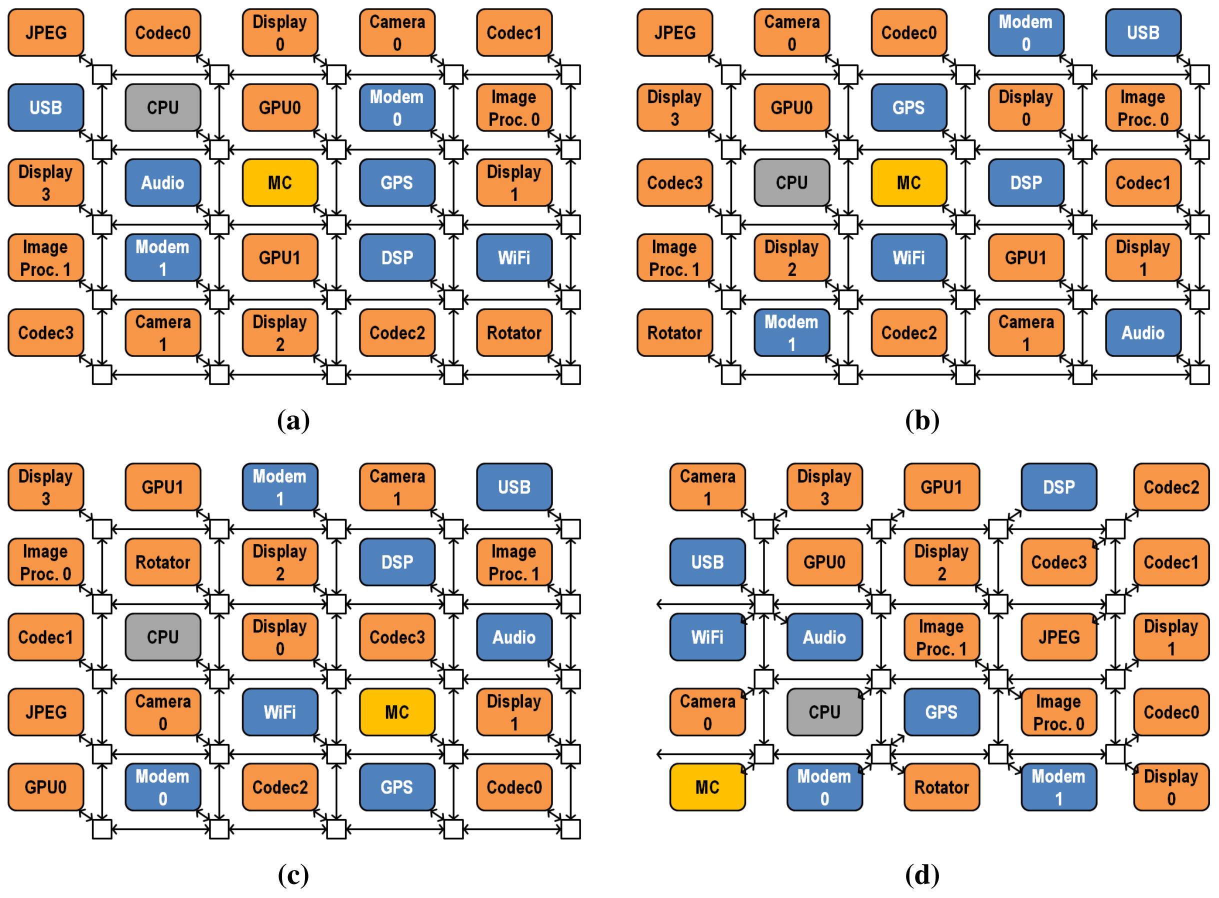
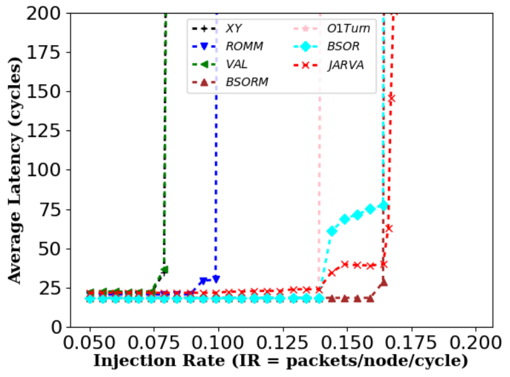


**Figure 4.3.** Summary of heterogeneous cores arranged in different 2D network configurations for a Camcorder application.

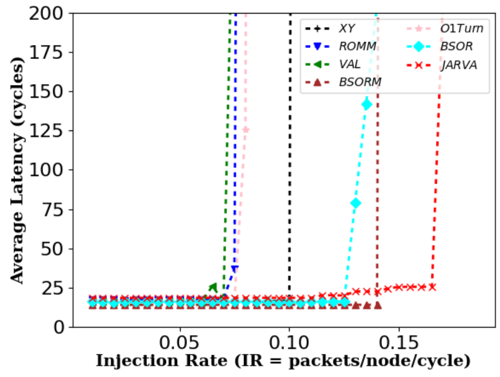
**Table 4.6.** Comparison of Maximum Channel Load on a Camcorder Application.

| Traffic | XY | YX | ROMM | VAL | O1TURN | BSOR | BSORM | JARVA | Δ |
|---|---|---|---|---|---|---|---|---|---|
| Camcorder(a) | 407 | 404 | 357 | 486 | 322 | 277 | 263 | 207* | 21.29% |
| Camcorder(b) | 241 | 294 | 233 | 476 | 264 | 241 | 259 | 207* | 11.16% |
| Camcorder(c) | 608 | 555 | 483 | 493 | 400 | 225 | 287 | 207* | 8.00% |
| Camcorder(d) | 553 | 681 | 500 | 614 | 424 | 258 | 447 | 219* | 14.70% |

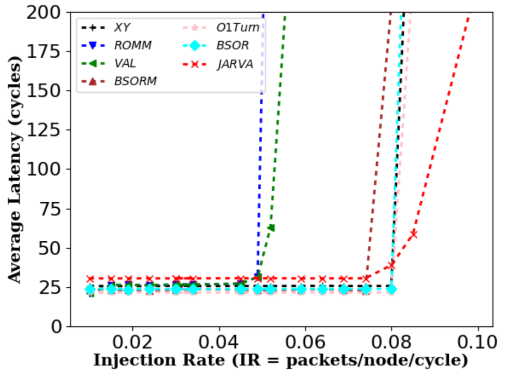
In this section, we present detailed flit-level simulations of the synthetic traffic and camcorder benchmarks described in Sections 4.4.2 and 4.4.3, respectively. We use the NOXIM [90] simulator, a flexible, highly configurable and a cycle-accurate network-on-chip simulator. We estimate the latency statistics of the injected flows for various oblivious routing algorithms discussed in Sections 4.4.2 and 4.4.3. The simulator is configured to have a per-hop latency of one cycle, four virtual channels (VC) per port, and the buffer depth of 16. We configure the simulator with an $8 \times 8$ 2D mesh network for the synthetic benchmarks and a $5 \times 5$ 2D mesh network for the camcorder benchmarks. The packet sizes (number of flits) for each source destination pair are consistent with our experimental settings of Table 3 and Table 4. To measure the latency statistics, we inject the packets using the injection rate parameter. For each simulation, the network is warmed for 10,000 cycles and the latency statistics are collected for 500,000 cycles after the warm-up. In our latency curves, the injection rate on the horizontal axis indicates the likelihood of packet injection per clock for each source node. For example, when we set the injection rate to 0.01, a packet (from a source node) is injected every clock cycle with a probability of 0.01. For a reasonably large simulation time (500,000 cycles in our experiments), this corresponds to an average of one packet for every 100 cycles. As the injection rate is increased, the network is expected to reach a saturation point and the latency shoots up. We seek to validate if lower MCL values of `JARVA` can push the saturation point rightwards, indicating that the network can withstand higher injection rates as compared to the previous works. The plots in Figure 4.4 support our hypothesis that smaller MCL of `JARVA` pushes the latency curves rightwards. Similarly, the plot of Figure 4.5 demonstrate the latency statistics for camcorder applications. Improved injection rates (for network saturation) are a manifestation of improved MCL from our `JARVA` formulation.



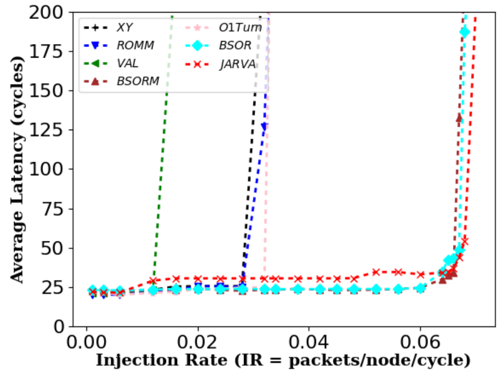
**(a)**



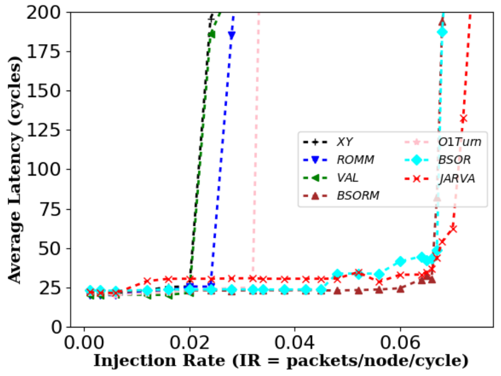
**(b)**



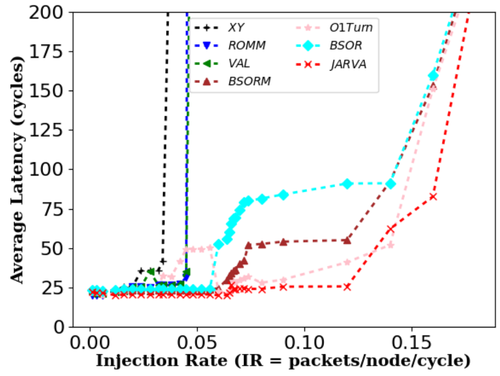
**(c)**



**(d)**

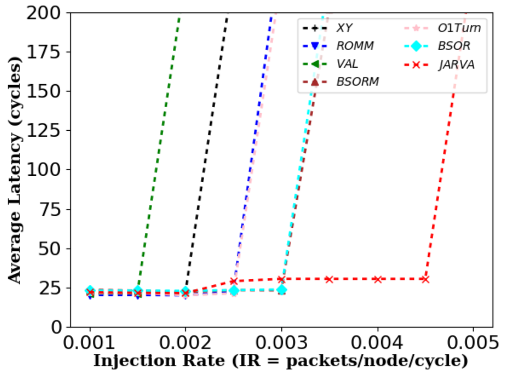


**(e)**

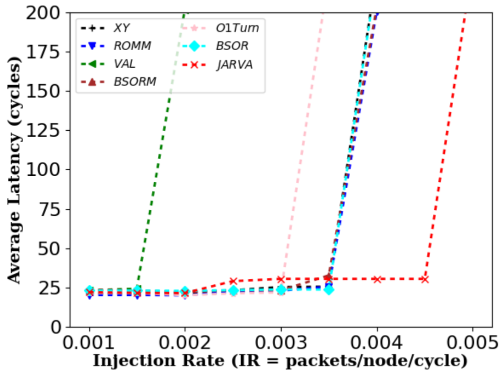


**(f)**

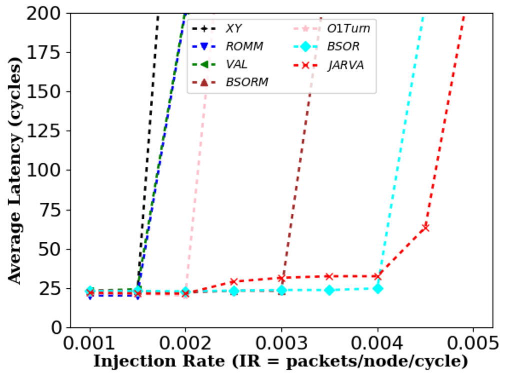
**Figure 4.4.** Latency as a function of injection rate for various routing algorithms on (a) Transpose (b) Shuffle (c) BitComp (d) Hotspot1 (e) Hotspot2 and (f) Hotspot3 patterns.



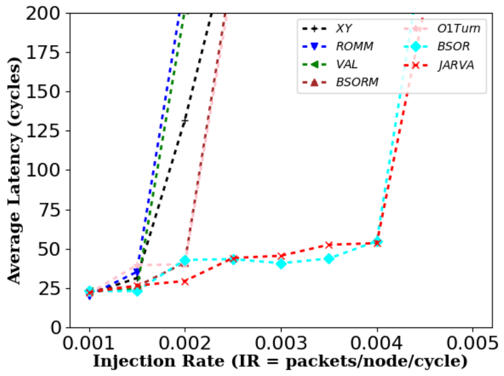
**(a)**



**(b)**



**(c)**



**(d)**

**Figure 4.5.** Latency as a function of injection rate for various routing algorithms on the camcorder(a-d) applications.

## 4.5 Conclusion

In this paper, we propose an SMT-based framework for joint application-aware oblivious routing and static virtual channel allocation `JARVA`. By jointly solving both problems together, we avoid unnecessary routing restrictions imposed by previous work. In our joint optimization approach, any path can be used to route a flow as long as there is a corresponding VC assignment that avoids deadlocks. Our SMT formulation of the problem allows us to succinctly capture the complex interactions between demand-sensitive routing and deadlock free VC assignment. Our evaluations show that our approach can achieve up to 30% better performance than the state-of-the-art application-aware oblivious routing algorithms and substantially better still in comparison with conventional oblivious routing approaches.

Chapter 4 contains materials from “Joint Application-Aware Oblivious Routing and Static Virtual Channel Allocation”, by Uday Mallappa, Chung-Kuan Cheng and Bill Lin, which appears in IEEE Embedded Systems Letters, May 2022. The dissertation author was the primary investigator and author of this paper.

# Chapter 5

# AI Acceleration: Skipping Ineffectual Computations

## 5.1 Introduction

A plethora of studies have aimed to alleviate the computation and energy cost of deep neural networks (DNNs) by hardware-friendly algorithmic innovations such as weight or filter pruning [91, 92, 8], and weight or activation quantization [93, 94, 95, 96, 97]. Some hardware techniques skip the operations with ineffectual operands, i.e., zero weights and/or input activations[1] [6, 43, 98]. Other studies even exploit bit-granular sparsity in the weights or inputs by breaking the operations into bit level and skipping if either operands' bit is zero [99, 100].

The sparsity-aware accelerators aim to skip the zero weights and/or inputs, where the zero inputs are produced by the preceding ReLU layer (Fig. 5.1). Skipping the ineffectual outputs is more beneficial than the ineffectual inputs as the number of zero outputs is more than zero inputs (most of the zero outputs are naturally discarded from the next layer's inputs after passing through the pooling). Several studies opt to skip the computation of negative outputs. The majority of these works split the input or weight bits into two significant and insignificant parts [101, 102, 103, 104]. The result of computation using the significant bits creates a mask that reveals which outputs need full-bit computation. Instead of splitting into bits, the work in [105] partitions the sorted weights into small groups and selects a representative weight from each one.

[1]We will refer to input activations as *inputs*, and output activations as *outputs*.

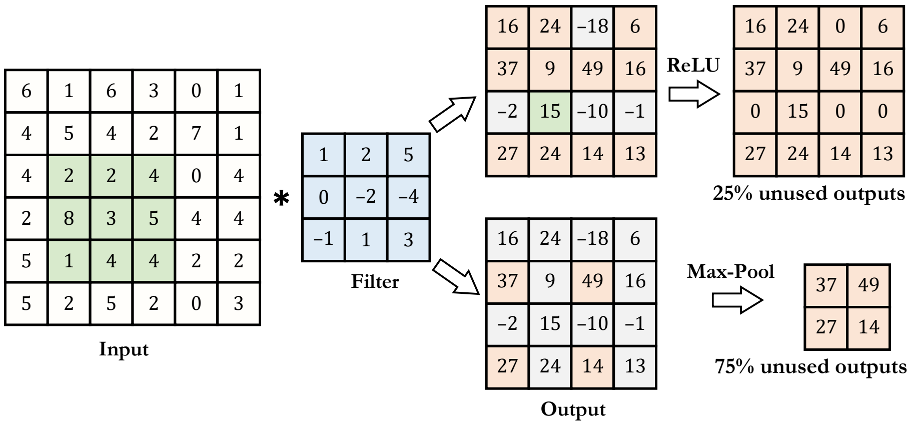


**Figure 5.1.** Ineffectual outputs due to ReLU and max pooling.

The result of convolution (dot-product) with these select weights and the corresponding inputs determines the potentially negative outputs.

In these studies, the prediction procedure, i.e., splitting of bits or partitioning the weights, is determined *statically* while the inputs' statistics can vary significantly from image to image (e.g., [97] points out this observation and proposes dynamic quantization based on input sensitivity). Thus, such prediction approaches have to be pessimistic to retain the accuracy, which results in low performance gain, e.g., $\leq 30\%$ in relatively complex networks [105, 103, 104]. Better improvement ($\sim$60%) is reported in [102]. However, [102] uses non-complex networks as well as high bitwidth for the weights and activations, which make the negative output prediction simpler. Also, when the bitwidths are already low, the impact of LSB bits becomes substantial, i.e., the likelihood of output sign change (misprediction) by continuing the operation on the four LSB bits of 8-bits weights is higher than eight LSB bits of 16-bits weights (as $2^{-5} + \cdots > 2^{-9} + \cdots$).

In this work, we propose algorithmic innovation and hardware support, `TermiNETor` for *dynamically* predicting and skipping the ineffectual outputs, including negative outputs preceding the ReLU layers and non-max outputs preceding the max-pooling layers. To this end, we leverage bit-serial operation whereby a convolution, represented as dot-product of flattened activations and weights $'A \cdot W'$, is split to $\sum A \cdot (W_i \times 2^i)$ over the weight bits $W_i$. After processing an $i^{\text{th}}$ bit of the weight vector, `TermiNETor` checks if the *partial output*[2] falls behind a designated threshold. This threshold varies for bit indexes; partial output produced using the initial (MSB) weight bits needs a more conservative threshold to ensure safe termination since the probability of a negative-to-positive flip of the output is higher as we move from MSB towards the LSB. The threshold also varies between the layers of a network.

There are three principal differences between `TermiNETor` and previous works: (i) `TermiNETor` uses bit-slicing and can flexibly terminate the operations at any index upon prediction, and notably, (ii) `TermiNETor` *operates in the granularity of output level*, meaning that it can terminate the operations for *a small group of output pixels* independent of the others, and (iii) Computations used for mask generation are reused for effective or useful convolutions; obliterating the computation overhead of standalone mask generation. Nevertheless, realizing a non-pessimistic threshold mechanism that universally works for various inputs is challenging. The algorithmic contribution of `TermiNETor` is to make such a partial-output-based prediction mechanism viable. Moreover, architectural support to take utmost performance and energy gain of these fine-grained terminations with high resource utilization is as critical, which is the focus of `TermiNETor`'s hardware novelty.

In the rest of this paper, in Section 5.2 we elaborate the `TermiNETor` algorithm for ineffectual output skipping preceding the ReLU and max-pooling layers as well as calibrating the model to avoid accuracy loss. In Section 6.3, we expound `TermiNETor` architecture to implement bit-serial-based output prediction and effectively utilize the terminated processing elements to improve the performance. In Section 6.4, we extensively evaluate `TermiNETor`'s algorithm and hardware by comparing with related works in terms of effective operation reduction (algorithm) and performance/energy (hardware). Finally, we conclude the paper in Section 7.1.

[2]A *partial output* is the dot-product of the whole activation vector and sub-bits of the whole weight vector (while a *partial sum* refers to *sub-vectors* dot-product result).

## 5.2 Ineffectual Output Skipping

Convolutional layers are the core building block of DNNs and contribute to $> 99\%$ of these networks computations [106]. These layers are followed by a nonlinear activation function, which has significant impact on the accuracy of these networks. The rectified linear unit (ReLU), simply defined as $f(a) = \max(0, a)$, is the widely-used and most successful activation function and yields better and more consistent performance than others such as hyperbolic tangent or augmented alternatives like leaky ReLU [107]. The ReLU function introduces sparse representations, i.e., zero output activation values, into the network and steers the observation that computations spent for the convolutions that generate negative outputs are unnecessary or *ineffectual*.

Besides the convolution and nonlinear activation layers, a downsampling operation is used intermittently to reduce the dimensionality of the output activation map. The downsampling is achieved via either a patch-wise (or window-wise) max-pooling or an average-pooling layer. In the former case, the pooled activation map highlights the largest feature of the patch and has been found to work better than average pooling for computer vision tasks like image classification [108]. In a $2 \times 2$ patch, three out of the four activations are discarded through the pooling operation. Therefore, another key observation is that 75% of the computations spent for the convolution operations that are preceded by a max-pooling layer are ineffectual.

Fig. 5.2 shows the percentage of ineffectual outputs for different layers of the VGG16 network on CIFAR-100 images. In this case, ineffectual outputs account for 61.8% of the produced outputs. The figure also shows the number of predicted ineffectual outputs by `TermiNETor` after using the first three bits (towards MSB) of the weight. Certain convolution layers are followed by a max-pooling layer, wherein $> 80\%$ of the layer outputs become ineffectual. The percentage of ineffectual outputs increases with the image size since a major portion of such images consists of a background, which is filtered out by the convolution. For instance, in the same VGG16 network, 77.4% of the outputs become ineffectual for $224 \times 224$ ILSVRC-2012 images. We

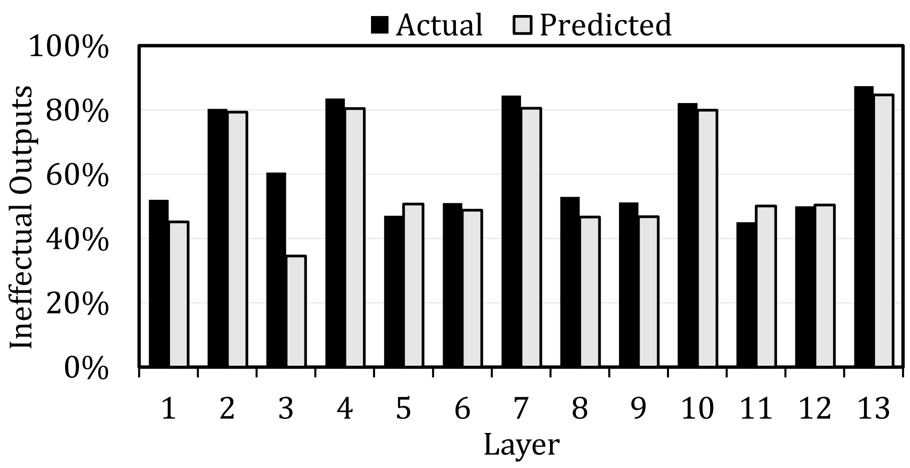


**Figure 5.2.** Ineffectual outputs of VGG16 on CIFAR-100 images.

exploit the presence of ineffectual outputs which are useless for the next layer's operations to improve the inference performance by skipping the computations that lead to such outputs. This is accomplished by predicting the ineffectual outputs using the proposed *weight-bit-serial* `TermiNETor` framework and architectural support to skip them with the fine granularity for maximal performance gain.

### 5.2.1 Inference Framework

In `TermiNETor`, the weights and input activations are quantized to standard 8-bits INT8 representation, bit 7 (MSB or sign) to bit 0 (LSB). Nevertheless, our bitwise prediction can be indeed employed with any $>$ 1-bit quantized model (though the efficiency gain will vary). Note that previous work such as [102] use over-provisioned bitwidths (e.g., 16) which makes the prediction much simpler. For instance, they can split the activations and/or weights into two 8-bits pieces and accurately predict the output sign by using 8-bits of each (75% effective operation reduction) since the lower 8-bits, in most cases, only adjusts the precision of the output rather than flipping the sign. In contrast, `TermiNETor` operates on the weights *bit by bit* and uses the complete 8-bit activations for each weight bit. Generating one convolution output begins with the MSB weight bit (bit 7) and continues till bit 0 of the weight.

Since only one bit of weights are processed at a time, the partial output is computed using the typical shift-and-add multiplication similar to bit-serial-based works [99, 100]. Regardless of output skipping, an advantage of such an architecture over conventional fixed-point designs is supporting arbitrary and layer-wise weight quantization [109]. A typical convolution takes eight iterations. The key idea is to use the partially computed outputs after $C$ cycles of the bitwise shift-and-add operations and predict (i) if the final output activation is leaning towards a negative value, and (ii) if the convolution output ends up unused because of a succeeding $2 \times 2$ max-pooling window. Accordingly, if we predict an ineffectual output at the $C = 3^{\text{rd}}$ cycle (i.e., after bit index 5), we can save five cycles by terminating the subsequent bitwise computations. Note that consuming three bits for prediction is principally different from 3-bits weight quantization. Unlike quantization, here, the model still operates with 8-bits weights for useful outputs and fewer bits are only used for ineffectual outputs. Our bitwise prediction can be employed with any $> 1$-bit quantized model (though the efficiency gain will vary). Fig. 5.3 depicts `TermiNETor`'s inference scheme. `TermiNETor` uses a masking operator to skip the convolutions that lead to ineffectual outputs. The mask is being updated by adding more skippable outputs after processing a new weight bit index.

**Formulation:** Consider the bitwise generation of a convolution output activation (pixel) $O^{p,q}$ at location $(p,q)$ of a certain output channel $M$ (being produced by filter $M$). Each convolution is multiplication of a three-dimensional filter with the corresponding part of the input feature map, which can be shown as a flattened dot-product (vector-vector multiplication). We start with the $i = 7^{\text{th}}$ (MSB) of the weight and build up the partial output using the shift-and-add operation (the first bit determines the sign). Notice that in INT8 representation, the weights are considered 8-bits integers.

$$O^{p,q} = A \cdot W[7:i] = \sum_{i=7}^{0} A \cdot (W^{(i)} \times 2^{i} \times -1^{i==7})$$

If $i \leq 8 - T$ (i.e., after processing $T$ weight indexes without skipping), we start masking

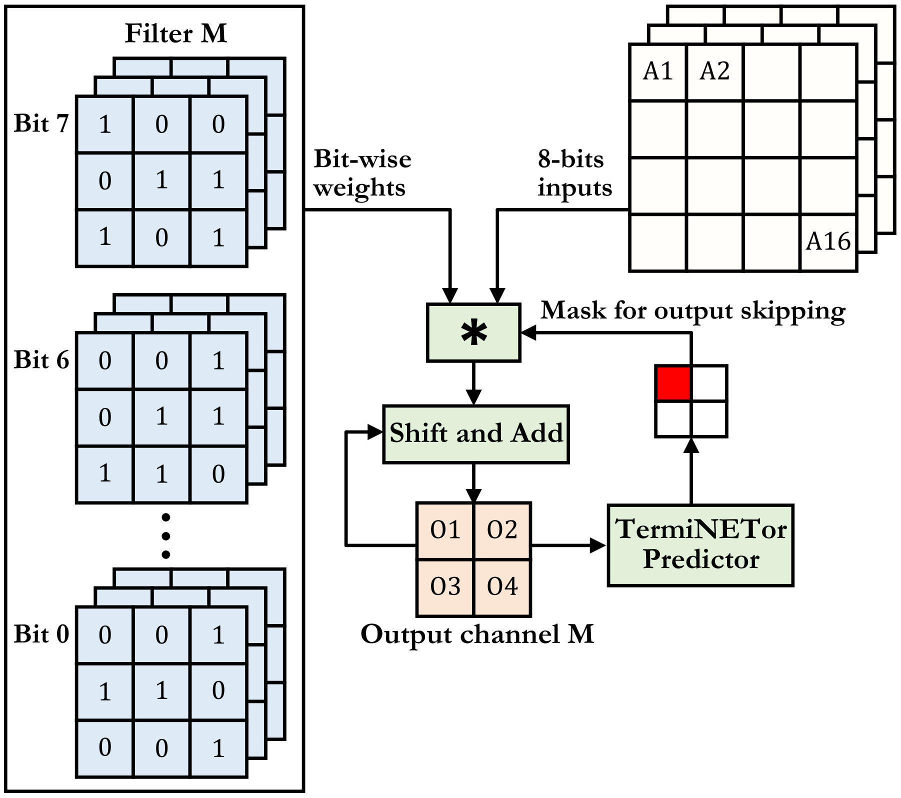


**Figure 5.3.** Early output prediction of `TermiNETor`.

the outputs that are predicted to be negative. Nevertheless, the partial output $O^{p,q}$ is incomplete and its sign might be changed if more indexes are processed. Fig. 5.4 demonstrates such a scenario. After the first bit, $A \cdot W[7:7]$ is negative but it increases and eventually its sign flips after using four bits ($A \cdot W[7:4]$).

We observed that adding a $\Delta^{p,q} \geq 0$ term that accounts for the residual convolution yields better prediction. We can think of $\Delta^{p,q}$ as an inexpensive positive bias term that is easy to compute and is also spatially aware of the input activation map. Thus, we have:

$$\text{if } i \leq 8 - T \text{ and } O^{p,q} + \Delta^{p,q} < \text{threshold} \Rightarrow mask^{p,q} = 1 \tag{5.1}$$

Essentially, $\Delta^{p,q}$ ensures that the partial output is negative enough to avoid potential sign change. Also for the convolution layers that are followed by a max-pooling, we mask the outputs that are

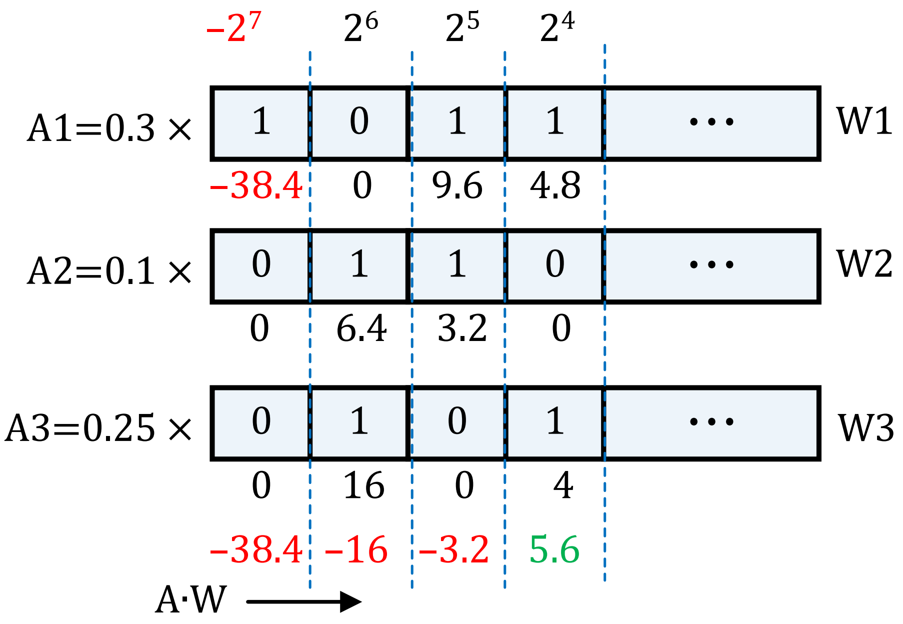


**Figure 5.4.** An example of negative-to-positive sign flip.

predicted to be ineffectual (i.e., are smaller than the maximum element of the pooling window $\mathscr{P}$):

$$\text{if } i \leq 8-T \text{ and } (p,q) \neq \text{argmax}\, \mathscr{P}_{(p,q)} \Rightarrow mask^{p,q} = 1 \tag{5.2}$$

### 5.2.2 Model Calibration

We observe accuracy degradation when the inference procedure of `TermiNETor`, explained above, is performed using a baseline pretrained model. This is because the pretrained model is not tuned for the imposed inference approximations; similar to, e.g., quantization wherein the weights require post-tuning. Also, the hyperparameters $T$ (that allows skipping after a particular weight bit index) and $\Delta$ (prediction bias) are layer-dependent and requires a rigorous design-space exploration.

To address the former issue, we calibrate the pretrained model to compensate the approximations imposed by `TermiNETor`'s inference, summarized by Algorithm 5. During the calibration process, the model tunes the weights toward the solution space that is highly sparse in the lower portions of the 8-bits weights. Such a weight calibration makes the lower portion of

**Algorithm 4.** `TermiNETor` calibration framework

```
Require: Pretrained weights $\Theta_0$, train_data = $\{X, y\}$
 1: model ← SGD($\Theta_0, X, y$)
 2: for each epoch in total_epochs do
 3:     for each layer $\ell$ in model.layers() do
 4:         for each bit_idx in $[7, \cdots, 0]$ do
 5:             if bit_idx $< 8 - T^\ell$ then
 6:                 if $\ell.index = 1$ then
 7:                     $Y^\ell$ += $\ell$.forward(X)
 8:                 end if
 9:                 if $\ell.index > 1$ then
10:                     $Y^\ell$ += $\ell$.forward($Y^{\ell-1}$)
11:                 end if
12:             end if
13:             mask$^\ell$ ← genMask($Y^\ell$, bit_idx, $\ell$.pool)
14:             if bit_idx $\geq 8 - T^\ell$ then
15:                 if $\ell$.index == 1 then
16:                     $Y^\ell$ += $\ell$.forward($X$, mask$^\ell$)
17:                 end if
18:                 if $l.index > 1$ then
19:                     $Y^\ell$ += $\ell$.forward($Y^{\ell-1}$, mask$^\ell$)
20:                 end if
21:             end if
22:         end for
23:     end for
24:     $\Theta_{t+1}$ ← SGD($\Theta_t$)
25: end for=0
```

the weights to be less impactful as compared to the upper portion. Notice that the calibration procedure is different from quantization. In quantization, cutting the discarded LSB bits does not affect the model accuracy. However, in our case, the LSB bits still play a major role to retain the model accuracy (which we further analyze in Section 6.4); *by calibration, only the probability of negative-to-positive sign flip is decreased.* Algorithm 5 processes the weight bits one by one (line 4) and updates the forward propagation outputs of the layer $\ell$ (denoted by $Y^\ell$) after each bit. If more $T^\ell$ or more bits are consumed, the algorithm applies the mask on the forward propagation values, which replaces the masked values with 0.

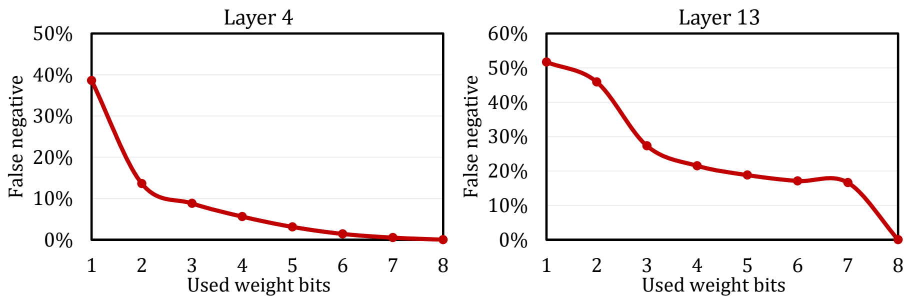


**Figure 5.5.** Percentage of false negative for two layers of a pretrained VGG16 model on CIFAR-100 dataset. The X-axis is the number of used weight bits.

## 5.2.3 Hyperparameters Exploration

**Minimum prediction index $T$:** In bit-serial processing of the weights, the confidence of outputs estimation improves as we move from the MSB bit towards the LSB bit. The key metric to evaluate the confidence of early termination is the *false negative* percentage, i.e., the percentage of outputs that are predicted negative but would turn to positive if all the weight bits are used. As shown in Fig. 5.5, the percentage of false negatives is layer-dependent, and it decreases as we consume more bits. Therefore, the value of $T$ is first derived from the plots of the pretrained networks, and then we use model calibration to mitigate the accuracy loss. For example, $T = 4$ might be acceptable for layer 4 (the left figure) but it can incur accuracy loss for layer 13.

**Mask generation and bias term $\Delta$:** The partial output prediction mechanism uses a simple addition and comparison. It takes the partial output sum, adds a pre-computed value $\Delta$, and checks if the result is greater than a desired threshold. The pre-computed value $\Delta$ *is unique for each output sum*, while the threshold is the same for all the outputs of a layer. Both the parameters are computed offline and are loaded for comparison during run-time. To ensure that minimum number of outputs will use all the weights' bits, the network is calibrated to make the weights' bits more sparse, i.e., more bits in a weight parameter are zero. This helps in obtaining a threshold value that gives a higher prediction confidence corresponding to each weight bit. As we showed in the example of Fig. 5.4, zero bits do not affect the partial output. At the same time,

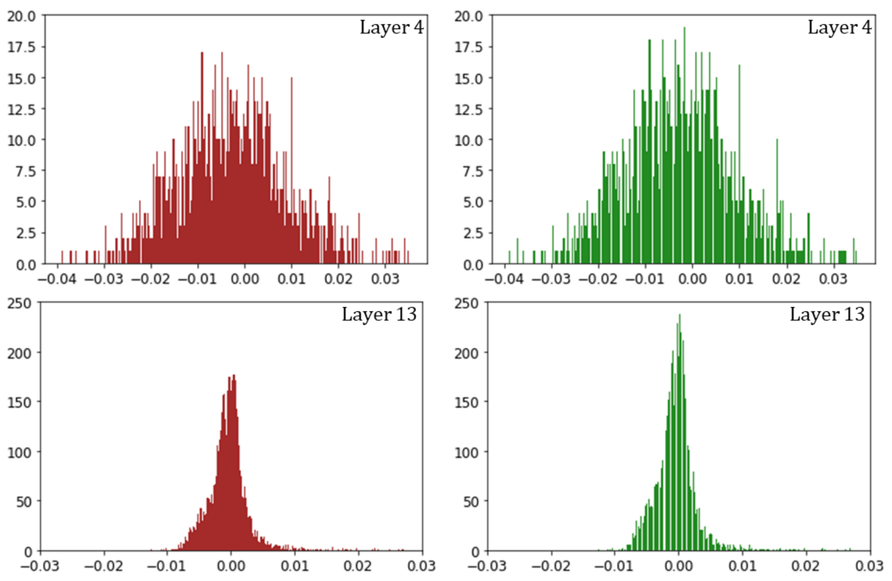


**Figure 5.6.** Weight histograms before model calibration (left) and after (right) for two layers of VGG16.

the more zeroes present in the weight, the higher the probability of terminating that computation earlier. Fig. 5.6 shows the impact of model calibration on the distribution of two representative layers of VGG16. The histogram is sparse around certain values which have large number of '1' bits as the frequency of such values is minimized in the calibrated model.

For every layer of the network, the input activations corresponding to the training data set are analyzed (offline) by taking their product with the kernel weights to determine an appropriate bias $\Delta^{(p,q)}$ for each output of the layer. After this, the threshold value (see Equation (5.1)) is obtained heuristically by analyzing the output predictions considering the obtained $\Delta^{(p,q)}$. We performed multiple experiments with the partial output prediction mechanism and realized that the prediction mechanism should not be employed as soon as the partial outputs are generated, since initially the confidence in the prediction is quite low and degrades the accuracy.

To improve hardware utilization, `TermiNETor` also implements group-termination, whereby processing a chunk of adjacent outputs (pixels with the same $(i, j)$ position but different channels) are terminated together. We elaborate group-termination in Section 6.3.

## 5.3 `TermiNETor` Architecture

### 5.3.1 Overview

The efficiency of the `TermiNETor` architecture is critical since, on the one hand, the processing elements should be able to operate on generating a new output upon early-terminating a particular output (rather than simply staying idle to simplify dataflow), and on the other hand, data reuse is a key efficiency factor which may contradict the first desideratum. This section elaborates on how `TermiNETor` can meet both criteria.

Fig. 5.7 shows the overview of `TermiNETor` datapath. It comprises a two-dimensional array of processing elements (PEs). The baseline architecture is an $8 \times 16$ array to consume a comparable power to previous works, but the architecture is scalable as we examine in Section 6.4. Each input lane broadcasts an input sub-image brick by brick to all the PEs in the same row (we use *lane* to distinguish between logical and physical memory entities). A *brick* (input, weight, or output) is composed of consecutive elements in the $Z$ axis, e.g., channels of an image. Similarly, each weight lane broadcasts a unique filter (occasionally, up to three different filters) at a time to all the PEs of the same column. Therefore, a PE row produces the same output indexes of different channels. The operations are bit-serial on the weights, so the weight lanes only transfer one bit of each weight. In our setting, a brick consists of eight elements. Therefore, each activation lane supplies $8 \times 8$-bits inputs, and each weight lane provides eight single bits (occasionally 16 or 24) of different weights (i.e., one bit of each). The activations and weights correspond to the same indexes; hence, the PEs can perform immediate MAC operations, which in practice is just a shift-and-add operation. The operations are orchestrated in a fashion that all PEs of a row terminate at the same time and move to the next sub-image, independent of the other rows, and only infrequent halts are needed to coordinate different PE rows. `TermiNETor` benefits from multiple levels of data reuse and sharing to reduce both on-chip and off-chip memory accesses, which we elaborate in subsection 6.3.3.

Once a weight brick is loaded (shared) into a column PEs, the PEs keep continuously

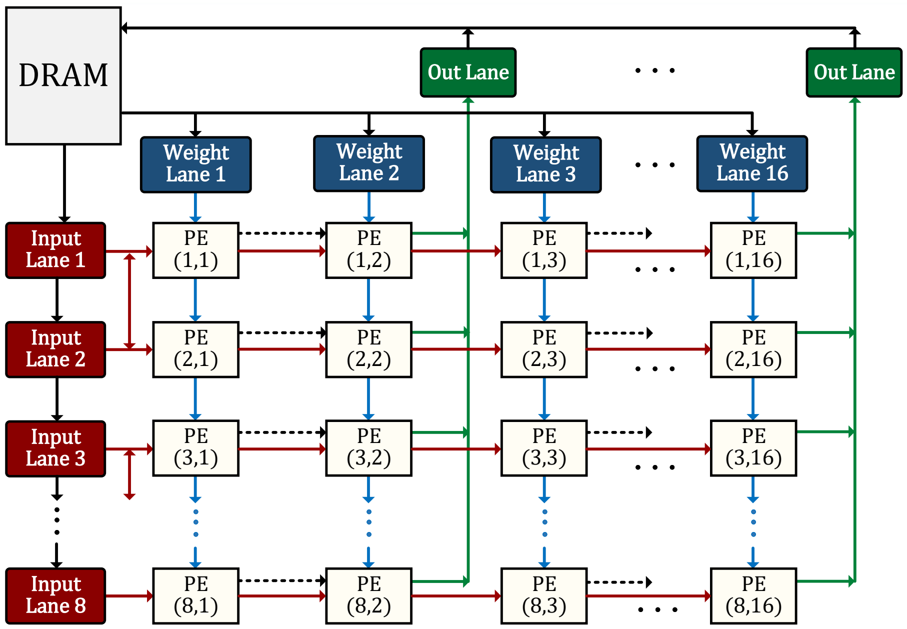


**Figure 5.7.** Overview of `TermiNETor`'s baseline architecture.

*reusing* this brick by fetching different activations, while at the same time the fetched activations are being *shared* and used among all PEs of a row. Also, after all the first (front) bricks of the current 16 filters are used (e.g., nine bricks for a $3 \times 3 \times Z$ filter), the front bricks of the *next batch of filters* are loaded, so they can operate on the same input activations stored in the input lanes. Thus, the inputs bricks will remain in the small on-chip memory until all filters use them. Since the filters are being processed bit by bit, the available weight lanes, on aggregate, can store one bit of all the filters in a given time, so `TermiNETor` does not need to evict and re-fetch a weight; all stored weight bits will be processed before evicting permanently.

## 5.3.2 Dataflow

`TermiNETor` partitions an $X \times Y \times \mathscr{C}$ input feature map into multiple $x \times y \times \mathscr{C}$ sub-images, as shown in Fig. 5.8. Each sub-image is stored in a separate input lane and is shared with all PEs of a row. The values of $x$ and $y$ depend on the PE's local register file (RF) size as well as the number of filters a PE receives. For instance, with 128 filters, each PE of the $8 \times 16$ array will visit $\frac{128}{16} = 8$ filters. Thus, an RF with a depth of 32 words can allocate four words per filter (i.e., $2 \times 2$ output pixels for eight channels); hence, the sub-images can be up to $4 \times 4 \times \mathscr{C}$ assuming filters of shape $3 \times 3 \times \mathscr{C}$ with sliding stride of 1. The number of sub-images can be more than the input lanes (PE rows). In that case, the remaining sub-images will be processed in multiple similar iterations after the currently stored sub-images are processed for all the filters. `TermiNETor` processes the stored sub-images as follows.

First, brick 1 of filters 1 to 16 are loaded into PE columns 1 to 16, respectively. As alluded above, a brick consists of eight consecutive elements in the $Z$ axis (channels). Operating at the brick level avoids local data storage and simplifies the compute elements. Initially, all PEs of a column share the same filter brick. The filters' bricks contain eight weights, but only one bit of each. Hence, only $W[7]$ of all weights are loaded at this stage. We pack the same index of the weights in the same memory word for regular memory accesses. The sub-images are scanned from the bottom row toward the top. Overlapping bricks of adjacent sub-images will be transferred from one input lane to the other. E.g., in Fig. 5.8, input lane 1 stores the $4 \times 4$ green sub-image (indexes 1 to 16); to produce output pixel index 3, it needs to read some of the inputs from the red sub-image. After applying brick 1 to the *corresponding* bricks of the input sub-image (i.e., to the inputs that need the brick 1 to generate the $2 \times 2$ output in the above example), bricks 2 to 9 of the current filters are applied to their corresponding input bricks, as well. Thereafter, we move to the next bricks (deeper channels) of the sub-image as well as of the loaded filters. Finally, after processing all the channels, we load the next batch of the filters (17–32) and repeat the same procedure.

So far, one bit of all the filters ($W[7]$) is used. Before repeating the whole procedure with $W[6]$, we examine if certain PEs can be released. We perform the PE release in the granularity of a row. All PEs of a row generate the same $(i, j)$ positions but for different channels. Since the outputs among the same positions of different channels are more correlated, we enforce this row-termination during the `TermiNETor` calibration. That is, all outputs produced with the PEs of a row try terminating at the same weight index, which we call row-termination or group-termination. In particular, the outputs window generated by a row is small (e.g., $2 \times 2$

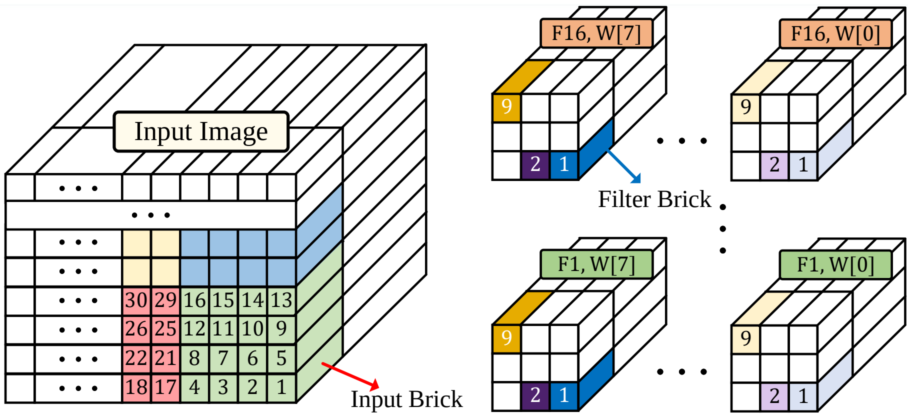


**Figure 5.8.** A sample input and filters for `TermiNETor` dataflow explanation.

when there are 128 filters, or $2 \times 1$ for 256 filters), which keeps the value correlated.

Once a row is terminated, its input lane loads another sub-image and starts over with $W[7]$ of filters 1–16 for the new sub-image. Nevertheless, the weight lanes are shared among all the rows. Therefore, the other rows might need to continue processing with $W[x \neq 7]$ and cannot supply $W[7]$. However, all the weight lanes supply the same filter at a given time. That is, the terminated row needs to load filters 1–16 with $W[7]$, and the rest of the rows need to also load filters 1–16, albeit weight $W[x \neq 7]$. Also, all the rows and PEs use the same brick index, which facilitates address generation. Therefore, at the cost of increasing the weight lane size, we use multiple banks so a weight lane can provide different weight bits when certain rows advance the other ones. It is straightforward as the brick index, and hence, the memory address will be the same for all rows. Note that the weight lanes are small as they store a single bit of a filter at a time. We limit the number of different indexes to three. On the rare occasions that more than three different weight bits are needed, we halt the outpaced rows to coordinate the rows.

The number of cycles to process an $X \times Y \times \mathscr{C}$ input image by the $8 \times 16$ `TermiNETor` array can be calculated as follows:

$$\text{cycle count} = \frac{X \times Y}{8} \times k_1 \times k_2 \times \frac{\mathscr{F}}{16} \times \frac{\mathscr{C}}{8} \times W_{\text{avg}} \tag{5.3}$$

where $k_1 \times k_2$ is the filters' kernel size (sliding stride of 1), $\mathscr{F}$ is the number of filters, and $W_{\text{avg}}$

is the average number of weight bits consumed among the outputs. The 8 in $\frac{X \times Y}{8}$ indicates the number of rows that can independently process a sub-image, 16 in $\frac{\mathscr{F}}{16}$ corresponds to the number of columns which enable loading the filters as batches of 16, and 8 in $\frac{\mathscr{C}}{8}$ denotes the brick size, since we process eight channels at once.

### 5.3.3 Data Reuse

`TermiNETor` takes advantage of different types of data sharing and reusing provided by its architecture and/or dataflow. Since a weight lane is shared among all the PEs within the corresponding column, a weight fetched from these memories will be used with multiple PEs of a column. The same also holds for the input activation lanes, i.e., the activations of a lane are broadcast to all 16 PEs of a row. In addition, a loaded weight brick stays in the PE until all the input activations that it can use are processed (e.g., in Fig. 5.8, after loading brick 1, all the sub-image inputs it can use are fetched in a row-major manner before loading brick 2). Furthermore, the sub-images that are stored in the input lanes are used for all filters before evicting, and there is no need to re-fetch from the DRAM. These activations remain in the input lanes until all the filters are processed. A difference between input bricks and weight brick is that a weight brick stays in the PE's local register, while new activations are constantly read from the local input lanes. Note that the input lanes consume significantly smaller power than DRAM.

Another type of filter reuse that `TermiNETor` supports is via batch processing. In the last convolution layers the image size becomes smaller, e.g., $4 \times 4 \times \mathscr{C}$. In such a case most of the PE rows become unused, but are still supplied with the filters that are shared across the columns. Hence, they can operate on different images using their independent input lanes. In batch processing mode, `TermiNETor` processes each image layer by layer before a layer $\ell$ causes PE underutilization. `TermiNETor` saves the inputs of the layer $\ell$ in the DRAM, processes other image(s) in the same fashion with full resource usage until layer $\ell$, and continues to process the layer $\ell$ and subsequent ones for the stored feature maps by loading multiple of them at a time. This helps to maximize PE utilization with a small increase of the embedded DRAM size as only

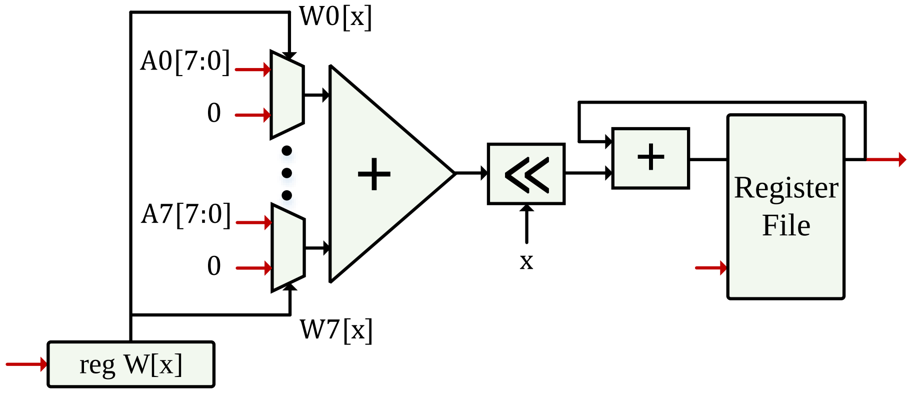


**Figure 5.9.** A processing element (PE) of `TermiNETor`.

one (and small) layer per image needs to be temporarily stored.

### 5.3.4 Processing Elements

`TermiNETor` benefits from uncomplicated processing elements, which is shown in Fig. 5.9. As explained in subsection 5.3.2, a brick of eight different weights, one bit of each, is loaded into the PE and is applied on all required bricks of the row sub-image. Therefore, an eight-bit register stores the $8 \times 1$ bit weights. At each cycle, eight 8-bits activations are read from the input lane and broadcast into the PEs. We only latch these activations at the output of the lane SRAM using $8 \times 8$ flip-flops which drive the whole row; hence, no input register is needed in the PE. The 32-words RF accumulates the partial sums associated with the PE (e.g., for eight filters, four outputs per each). The address of the RF row, input from the controller, depends on which of the assigned filters and outputs are being processed. The $0 \leq x \leq 7$ denotes the weight bit index that is being processed and determines the amount of shifts.

Upon termination signal after a certain weight bit index, the content of the RF is transferred to the output lanes. For implementation purposes (to make the RFs compact), we share a $4\times$ wide RF between four PEs. Therefore, as shown in Fig. 5.7, an output lane is shared between several columns, and the links between PEs in a row show writing the computation result in the shared RF (note that RFs use the same address).

**Table 5.1.** Accuracy and operation comparison of `TermiNETor` with baselines.

| | Network | Accuracy | | | | Operation (M) | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | Base | SeerNet | 4-bits | TermiNETor | Base | SeerNet | 4-bits | TermiNETor |
| CIFAR 10 | VGG16 | 91.65% | 91.30% | 90.82% | 91.52% | 314 | 262 (1.20×) | 157 (2×) | 187 (1.68×) |
| | ResNet18 | 93.81% | 93.38% | 93.02% | 93.56% | 555 | 499 (1.11×) | 277 (2×) | 384 (1.45×) |
| | ResNet50 | 94.58% | 93.20% | 93.24% | 94.08% | 1298 | 1103 (1.18×) | 649 (2×) | 885 (1.47×) |
| CIFAR 100 | VGG16 | 70.40% | 70.12% | 69.01% | 70.07% | 318 | 267 (1.19×) | 159 (2×) | 191 (1.66×) |
| | ResNet18 | 75.41% | 74.96% | 73.15% | 75.13% | 555 | 492 (1.13×) | 277 (2×) | 381 (1.46×) |
| | ResNet50 | 77.30% | 76.84% | 75.12% | 77.01% | 1298 | 1056 (1.23×) | 649 (2×) | 893 (1.45×) |
| Image-Net | VGG16 | 56.91% | 56.04% | 55.15% | 56.84% | 1256 | 1017 (1.24×) | 628 (2×) | 833 (1.51×) |
| | ResNet18 | 61.84% | 61.01% | 59.92% | 61.66% | 2222 | 1990 (1.12×) | 1111 (2×) | 1568 (1.42×) |
| | ResNet50 | 64.13% | 63.32% | 61.90% | 64.02% | 5194 | 4672 (1.11×) | 2597 (2×) | 3685 (1.41×) |

## 5.4 Experimental Results

### 5.4.1 General Setup

We implemented the algorithmic flow of `TermiNETor`, i.e., bit-level mask and threshold generation, early convolution termination, and calibration (including enforcing row-termination) using PyTorch. For training, we used SGD optimizer, momentum of 0.9 with weight decaying, and learning rate from 0.1 down to 0.0002 over 100 epochs. To recap, the weights and activations are quantized to eight bits and the convolution operation is implemented by bitwise shift-and-add operations. Using the pretrained model, we derive the layer-wise bit index $T$ at which the dynamic mask (for ineffectual output predictions) generation starts. Our experiments showed that the mask generation can start after consuming the first three bits (towards MSB) for most of the layers. For certain layers (e.g., layer 4 of VGG16 on CIFAR-100), we can start the mask generation after consuming the first two bits of weights.

We implemented the `TermiNETor` accelerator in SystemVerilog and verified its functionality using Modelsim. We used Synopsys Design Compiler O-2018.06 to synthesize the RTL code, and Cadence Innovus 2019 for placement and routing using TSMC 40 nm 0.9V library (using a combination of HVT and RVT) for the typical (TT) process corner. We used TSMC 0.8V high-performance memory compiler to generate the dual-port register file (for PEs) and single-port SRAMs (for input activation lanes, weight lanes, and output lanes). Thanks to the simplicity of the processing elements, `TermiNETor` could achieve an operating frequency of

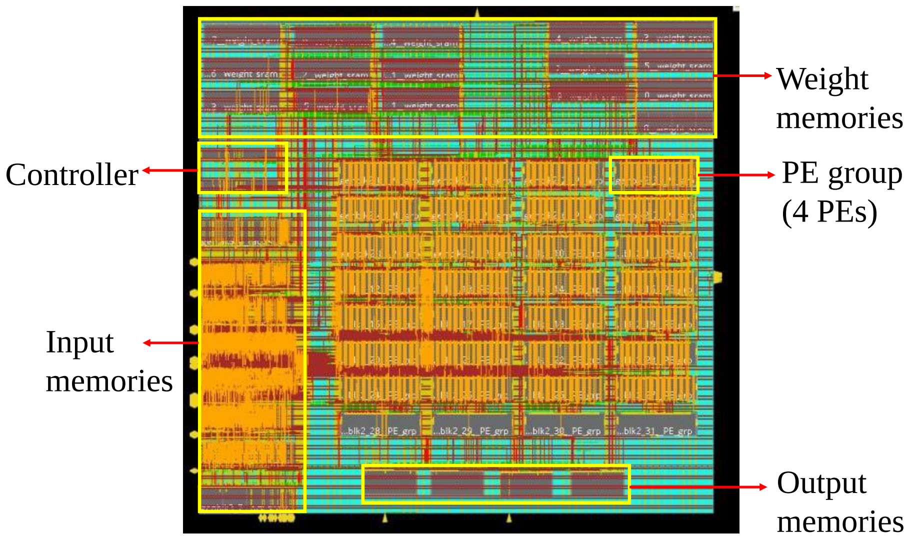


**Figure 5.10.** Physical layout of `TermiNETor` accelerator, implemented in TSMC 40 nm technology.

1 GHz. We used the post placement and routing netlist to report the area and power using Design Compiler. To calculate the DRAM access energy, we used Destiny [110] to generate a 32 MB (128 bit words) eDRAM model for 40 nm and added the DRAM power based on its aggregate read and write throughput. The 32 MB DRAM can store the whole input/output feature maps of any layer, as well as the model weights.

Fig. 5.10 illustrates the post placement and routing layout of the baseline `TermiNETor` ($8 \times 16$ array), which occupies 0.95 mm$^2$. Each of the eight rows are divided into groups of four PEs, that share a wide register file. The input activations memories (left), weight memories (top), and output memories (bottom) are also visible.

### 5.4.2 Operation Reduction

We evaluate the algorithmic effectiveness of `TermiNETor` by comparing it with SeerNet [103], which is a two-stage ineffectual output predictor. The first stage of SeerNet uses a 4-bits quanitized inference for mask generation, and then it uses the generated mask for full-precision (8-bit) inference as the second stage. The performance improvement, defined as effective operations, is reported with respect to the 8-bits baseline implementation (multiplication of a *w*-bits weight is considered as $\frac{w}{8}$ of a 8-bits weight multiplication). We also include the performance and accuracy metrics for the 4-bit weight quantization (which uses $\frac{1}{2}$ effective operations compared to 8-bits weights). In our evaluation, we considered VGG16, ResNet-18, and ResNet-50 networks using CIFAR-10 and CIFAR-100 datasets, as well as a 200-class subset of ImageNet (Tiny ImageNet).

Table 6.1 summarizes the accuracy and effective operation count for the aforementioned models and datasets. The *Base*, *SeerNet* and *4-bit* columns represent the baseline 8-bit quantized model, SeerNet inference [103] and 4-bit quantized models. SeerNet, which is only software-centric optimization, does not reuse the computations performed in the mask generation stage (4-bits inference), leading to computation overhead. This overhead in computation nullifies the gains of skipping ineffectual outputs when the output sparsity is less than 50%. It is worth noting that the results presented in SeerNet [103] do not account for the computations performed in the first phase (mask generation). In the following we summarize the results of different datasets.

**CIFAR-10:** As compared to the baseline VGG16 network, while SeerNet achieves 1.20× operation reduction with 0.35% accuracy loss, our method offers 1.68× reduction with only 0.13% accuracy loss. For residual networks such as ResNet-18, while SeerNet's speedup is 1.11×, our method offers 1.45× improvement with better accuracy. We observe a similar trend for ResNet50; 1.47× speedup in TermiNETor versus 1.18× of SeerNet, and 0.88% higher accuracy. Since down-sampling in residual networks is manly implemented using a stride-2 convolution, the sparsity of output activation is low, leading to low speedup values in the residual networks as compared to VGG16. Although the 4-bit model offers 2× speedup, it suffers accuracy loss of up to 1.34%.

**CIFAR-100:** For CIFAR-100, we achieve 1.66× speedup in operation count using VGG16, 1.46× using ResNet-18, and 1.45× on ResNet50, which are on average 34% higher than SeerNet, with 0.1% higher accuracy. The 4-bit quantization offers 2× speedup, but incurs 1.94% accuracy loss (versus 0.3% of our approach).

**TinyImageNet:** We observe a similar trend with the Tiny ImageNet dataset. Along with an improved operation reduction (up to 1.51× speedup) compared to the baseline, our

**Table 5.2.** Operation reduction with ~2% accuracy budget.

| | Network | Accuracy | | Operation (M) | |
|---|---|---|---|---|---|
| | | Base | TermiNETor | Base | TermiNETor |
| CIFAR 10 | VGG16 | 91.65% | 89.90% | 314 | 96 (3.27×) |
| | ResNet18 | 93.81% | 91.61% | 555 | 214 (2.59×) |
| | ResNet50 | 94.58% | 92.33% | 1298 | 496 (2.62×) |
| CIFAR 100 | VGG16 | 70.40% | 68.16% | 318 | 98 (3.24×) |
| | ResNet18 | 75.41% | 73.11% | 555 | 219 (2.53×) |
| | ResNet50 | 77.30% | 75.16% | 1298 | 503 (2.58×) |
| Image-Net | VGG16 | 56.91% | 54.94% | 1256 | 502 (2.50×) |
| | ResNet18 | 61.84% | 59.86% | 2222 | 958 (2.32×) |
| | ResNet50 | 64.13% | 62.12% | 5194 | 2144 (2.42×) |

improvements are 25.2% better than SeerNet with 0.72% higher accuracy (only 0.12% drop compared to the 8-bits baseline).

The 4-bits quantized models offer a constant 2× operation reduction. However, 4-bits models suffer an accuracy loss of up to 2.2%. Accordingly, we allowedTermiNETor to undergo an accuracy degradation of up to ~2.2% (similar to 4-bits models) by starting the mask generation early in the bit-serial processing of the weights as well as using more aggressive threshold values. With up to 2% accuracy loss as compared to the 8-bits baselines, we observe an operation reduction of upto 3.3× for VGG16 with CIFAR-10 dataset as compared to 2× speedup of the 4-bit model. As shown in Table 5.2, TermiNETor achieves better speedup values for other datasets and networks, compared to the 4-bit models.

**Table 5.3.** Characterization of TermiNETor components.

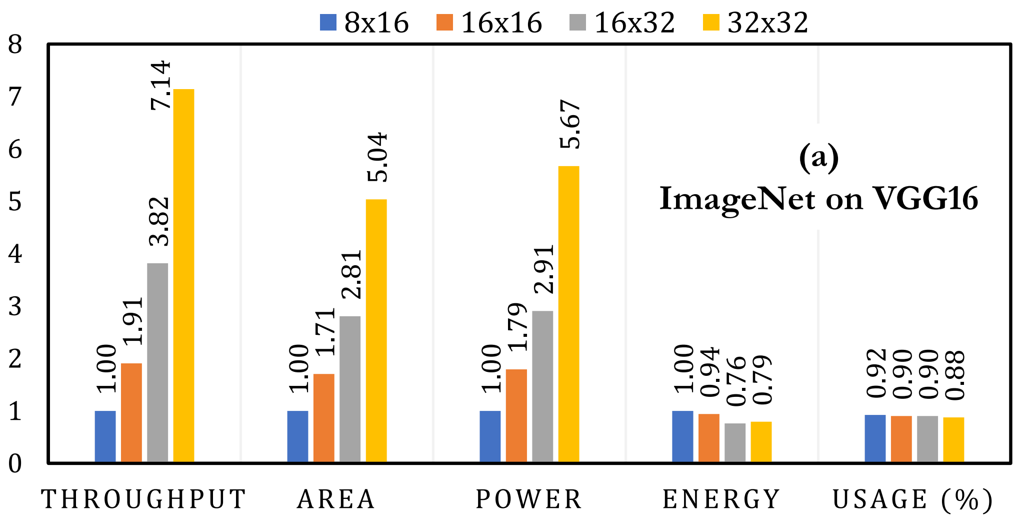
| Module (total) | Size/count | Area ($\mu m^2$) | Power (mW) |
|---|---|---|---|
| Processing Element | $8 \times 16$ | 81,263 | 42.8 |
| Activation Memory | 16 KB | 281,636 | 59.1 |
| Weight Memory | 18 KB | 271,596 | 15.2 |
| Output Memory | 4 KB | 75,163 | 1.9 |
| Register File | 10 KB | 233,987 | 78.2 |
| Controller | 1 | 9,258 | 4.78 |

### 5.4.3 `TermiNETor` Accelerator Evaluation

Table 5.3 presents the characterization of `TermiNETor` hardware components. Each PE has a $32 \times 20$bit register file, but as as mentioned in Section 5.3.4, we merge the RFs of four PEs in a row and share a $32 \times 80$bit RF among all. Considering 8-bits weights and activations, 20-bits words are sufficient to accumulate all partials (as most of the products cancel out each other). The entire array needs $32 \times 20\,\text{b} \times 8 \times 16 = 10\,\text{KB}$ of register file. To avoid DRAM stalls when flushing out the output activations from RFs, we use intermediate output lanes consisted of four SRAMs, each of which has a size of 1 KB ($256 \times$ 32-bits). We stall new computations in the architecture during the data transfer from the RFs to the output lanes, which takes $32 \times 8 = 256$ (an output lane is shared between all eight rows). In networks such as ResNet-18 and ResNet-50, this stall is utilized to perform residual layer addition.

The capacity of the activation memory enforces a limit on the number of input channels that can be stored in the buffer. Sparse outputs (as a result of early termination) demand more input activations to be stored in the activation memory because idle PE rows reduce the throughput. On the contrary, loading only a subset of input channels increases the DRAM accesses as the activations are re-fetched for every weight bit. With the sparsity distribution of VGG16, ResNet18 and ResNet50, our simulation experiments indicate that loading a maximum of 512 input channels into the activation memory balances the throughput and the DRAM accesses. In our architecture, we use eight activation memories, each with a size of 2KB ($256 \times$ 64-bit) transferring 64-bits to each PE row every cycle. Similarly, the weight memory supplies unique weight bricks to all the PEs in a column. To store the weights, we use 16 SRAMs (for 16 PE columns), each having 1152 unique eight bit words, with a total capacity of 18KB.

The $8 \times 16$ architecture of `TermiNETor` occupies a total area of $0.95\,\text{mm}^2$ (at 40 nm), consumes 4.0 mW leakage power, an average dynamic power of 222.5 mW (at 1 GHz), 9.2% of which is the DRAM access power (average among benchmarks). The baseline $8 \times 16$ array of `TermiNETor` is able to process 12.2 ImageNet-size ($224 \times 224$ images) implementing VGG16



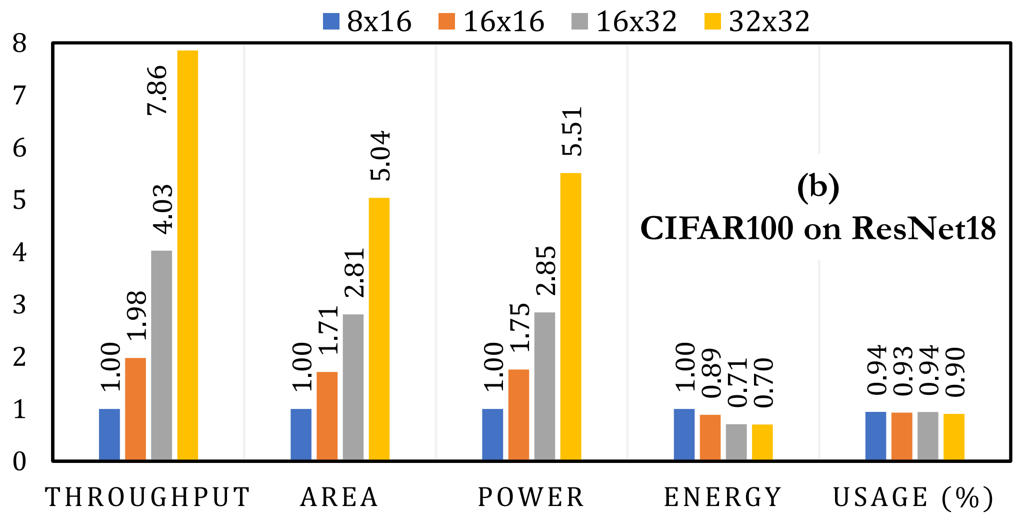


**Figure 5.11.** Scalability of performance, area, power, and utilization of `TermiNETor` running different image sizes.

(large image and network), or 235 smaller $32 \times 32$ images running ResNet-18 (small image and network). Fig. 6.6 shows how the throughput scales by increasing the `TermiNETor` accelerator size up to $32 \times 32$ array. The baseline is shown by $1.0\times$ (which denotes 12.2 image/second for VGG16, and 235 image/second for ResNet-18 as mentioned). The largest array, i.e., $32 \times 32$, which consists of $8\times$ more PEs, can increase the performance by up to $7.86\times$, which is almost linear with the number of PEs added.

The same Fig. 6.6 also shows the area sociability, which is independent of image size. While the baseline area is 0.95 mm$^2$ (denoted by $1.0\times$), the area of the largest array ($32 \times 32$)

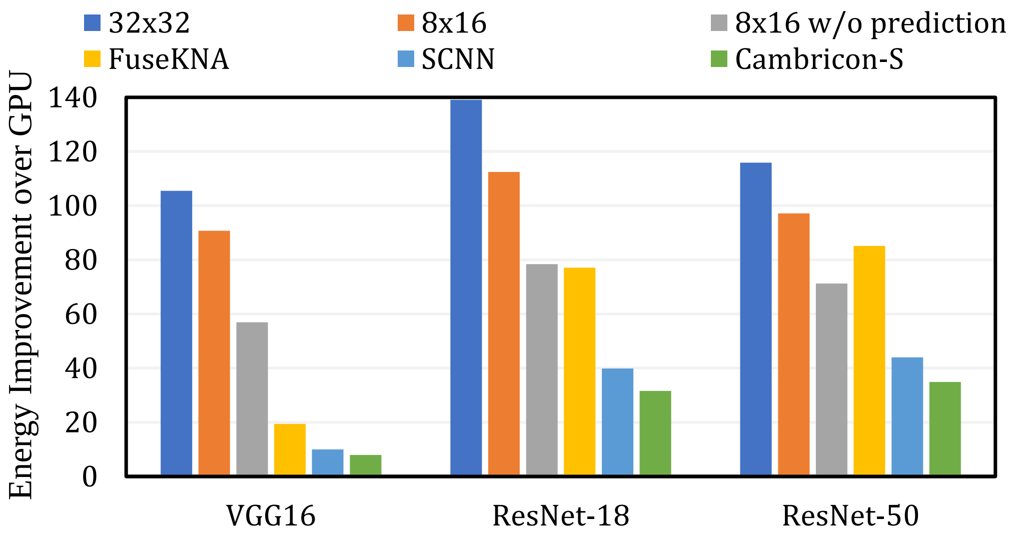


**Figure 5.12.** Energy per classification, normalized to GPU.

is only 5.04× larger (i.e., $5.04 \times 0.95\,\text{mm}^2 = 4.79\,\text{mm}^2$). The area increase is $< 8\times$ because although the PE count is increased by 8×, the number of input lanes of the $32 \times 32$ array is only 4× of the baseline array, and the number of weight and output lanes increases by just 2×. A similar trend can be observed for the power consumption. As alluded above, the leakage and dynamic power of the baseline array is 4.0 mW and 222.5 mW, respectively. The energy consumption reduces by increasing the array size due to better data reuse, i.e., the same weights and inputs are shared among more PEs, and a result, SRAM accesses are reduced. PE utilization is lower for ResNet-18 than VGG16 due to the residual layers, which have a $1 \times 1$ kernel that enables the output lane to process more filters at the expense of activations. For residual layers with larger number of input channels ($\geq 512$), there are a limited number of activations to start with, and early termination combined with the low capacity of the output buffer diminishes it even further. It results in a PE rows remaining unused for a significant time.

## 5.4.4 Comparison with Previous Work

Since the performance and power consumption depends on the resources (e.g., PE array size), we compare `TermiNETor` with state-of-the-art works in terms of energy consumption per classification. Particularly, the results of subsection 5.4.3 showed that the performance of

TermiNETor scales well with the PE array size; hence, high performance targets can be meet if the energy consumption is satisfactory. Therefore, we compare the energy consumption of TermiNETor (for both baseline $8 \times 16$ and larger $32 \times 32$ array, as well as the baseline baseline $8 \times 16$ *without* early termination) with SCNN [6], Cambricon-S [98], and FuseKNA [7]. The former two (SCNN and Cambricon-S) are zero weight/input skipping accelerators without bit-serial operations (i.e., sparsity-aware 8-bits accelerators), while FuseKNA skips zero inputs as well as repetitive computations using bit-serial operations. The results of SCNN, Cambricon-S, and FuseKNA are presented in [7] as normalized to Nvidia 1080 GTX GPU; hence, we repeated the experiments using the same GPU and normalized the TermiNETor results to the obtained result of GPU.

Fig. 5.12 compares the energy usage of TermiNETor and the other accelerators running $224 \times 224$ images on VGG16, ResNet-18, and ResNet-50, normalized to Nvidia 1080 GTX Ti GPU. The $32 \times 32$ array of TermiNETor achieves the highest energy efficiency and reduces the energy consumption by 120.1× over GPU, 1.98× over FuseKNA, 3.84× over SCNN, and 4.84× over Cambricon-S. Compared to the baseline $8 \times 16$ array with (without) early termination, the $32 \times 32$ array is 1.20× (1.75×) more energy efficient (with ∼5.7× higher power consumption according to Fig. 6.6, i.e., ∼1290 mW versus 226 mW of the baseline TermiNETor array).

## 5.5 Conclusion

In this work, we introduce TermiNETor framework to accelerates CNN inference by performing dynamic convolution termination for ineffectual output activations. During the weight bit-serial inference data flow, at every weight bit, we predict ineffectual output activations resulting from the ReLU and max-pooling layers. Using these predictions, the donwstream bitwise shift-and-add convolution operations are continued only for the useful output activations. We evaluate TermiNETor across various networks and datasets, demonstrating a significant reduction in operation count (up to 1.7× speedup) with negligible loss of accuracy. We also propose a novel accelerator that exploits the dynamic bitwise convolution terminations with an average energy efficiency of 120$\times$ as compared to GPU, and at least $1.98-4.84\times$ with respect to the state-of-the-art sparsity-aware accelerators. In the next chapter, we propose another neural network accelerator that reuses redundant computations during the inference phase. In addition to reducing the compute and energy burden, we also focus on reducing the memory footprint of the model, leading to an area-optimized silicon implementation.

### 5.5.1 Acknowledgements

This work was supported by TSMC, in part by CRISP, one of six centers in JUMP (an SRC program sponsored by DARPA), SRC Global Research Collaboration (GRC) grant, and NSF grants #1911095, #1826967, #2100237, and #2112167.

Chapter 5 contains materials from "TermiNETor: Early Convolution Termination for Efficient Deep Neural Networks", by Uday Mallappa, Pranav Gangwar, Behnam Khaleghi, Haichao Yang and Tajana Rosing, which appears in International Conference on Computer Design (ICCD), October 2022. The dissertation author was the primary investigator and author of this paper.

# Chapter 6

# AI Acceleration: Patterned DNN

Weight clustering is an effective technique for compressing deep neural network (DNN) memory by using a limited number of unique weights and low-bit weight indexes to store clustering information. In this chapter, we propose `PatterNet`, which enforces shared clustering topologies on filters. Cluster sharing leads to a greater extent of memory reduction by reusing the index information. `PatterNet` effectively factorizes input activations and post-processes the unique weights, which saves multiplications by several orders of magnitude. Furthermore, `PatterNet` reduces the add operations by harnessing the fact that filters sharing a clustering pattern have the same factorized terms. We introduce techniques for determining and assigning clustering patterns and training a network to fulfill the target patterns. We also propose and implement an efficient accelerator that builds upon the patterned filters. Experimental results show that `PatterNet` shrinks the memory and operation count up to 80.2% and 73.1%, respectively, with similar accuracy to the baseline models. `PatterNet` accelerator improves the energy efficiency by 107× over Nvidia 1080 1080 GTX and 2.2× over state of the art.

## 6.1 Introduction

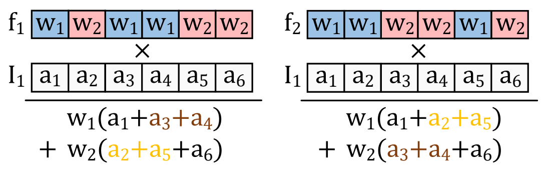
The ever-increasing efficacy of DNNs in diverse application domains is coupled with the increase in the size and computations of their models [111]. Extensive research has been done to alleviate the memory and computational burden of DNNs. Primary compression techniques include weight quantization [93, 94, 95], pruning [91, 92], clustering [92, 112], and filter pruning [113, 8], especially with a slant toward hardware efficiency such as hardware-aware quantization [96] and structured pruning [114].

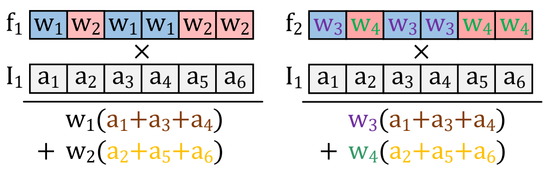
In weight quantization, the network parameters take values from a set of predetermined values (e.g., $-2^{k-1}$ to $2^{k-1}-1$ in uniform quantization), while weight clustering groups the weights into abstract clusters, where all weights of a cluster share the same value. Thus, by clustering we can simply store the cluster index/id of each weight (in *index table*), along with a small table that maps the indexes to weight values. Previous works [95, 115] show that $\sim$16 unique weights can retain the accuracy, which results in $2\times$ memory compression by storing $\log_2 16 = 4$-bit indexes instead of the primary 8-bit weights.

As shown in Figure 6.1, the convolution operation in CNNs is essentially a window-wise dot-product between a multi-dimensional filter and the input activations to generate output feature maps. Since clustering uses a limited number of unique weights, it can be leveraged for computation efficiency by factorizing weights. The example of Figure 6.1(a) shows filters clustered with two unique weights $w_1$ and $w_2$. Clustering can reduce multiplications (MULs) by first accumulating the inputs based on the weights clusters and applying MULs on the sum of factorized terms. For a filter with $n_w$ weights (typically $O(10^3)$), the number of MULs reduces from $n_w$ to **G** (the number of unique weights or clusters), where usually **G**=16 unique weights is sufficient as alluded earlier.

Factorization also results in common sub-groups of inputs. In Figure 6.1(a), both filters $f_1$ and $f_2$ computations have overlapping sub-groups $a_3 + a_4$ and $a_2 + a_5$. A previous study, UCNN [115], attempts to form compound sub-computations to be reused among multiple filters. Nevertheless, UCNN achieves less than 30% energy improvement due to the complexities involved in dealing with fine-grained sub-groups. FuseKNA [7] uses weights in a bit-serial fashion to slice MULs into ADDs. When processing each bit of multiple filters, FuseKNA reuses the overlapping ADDs among kernels.

In this paper, we take an unorthodox approach to increase computation reuse and reduce





**(a)** Factorization and sub-compute reuse between filters. **(b)** Whole *cluster* reuse among different filters.

**Figure 6.1.** Weight clustering with (a) computation reuse and (b) cluster reuse between filters.

memory by enforcing filters to share the same clustering pattern. Filters $f_1$ and $f_2$ in Figure 6.1(b) share the same clustering. That is, a particular weight at index $i$ of both $f_1$ and $f_2$ belong to the same cluster. This is distinguished by using the same background colors for clusters of $f_1$ and $f_2$. However, unlike baseline clustering [92] that uses the same unique weights, in `PatterNet` each filter can have a different set of **G** unique weights; hence, *filters share clustering patterns*, not exact data. As a result of pattern sharing, along with the reuse of whole activation groups between $f_1$ and $f_2$ (e.g., $a_1 + a_3 + a_4$ is repeated for both filters), the same cluster-index information can be used for both filters. Therefore, $f_2$ only needs to store its unique weights set (which is negligible compared to the eliminated index information), and carry out only **G** MULs on the pre-computed input sub-groups that have already been accumulated when processing $f_1$.

Our main contributions are as follows. In Section 6.2, we explore the potentials of patterned filters, propose a mathematical formulation to identify the patterns, and a training strategy to enforce the desired patterns without deteriorating the model accuracy. To the best of our knowledge, this is the first work that introduces patterned filters to save memory and computation of DNNs. In Section 6.3, we elaborate the dataflow, architecture, and processing units of `PatterNet` accelerator that supports networks with both patterned and conventional weight clustering. As weight quantization is a special case of clustering, our architecture supports quantized networks, as well. In Section 6.4, we examine the efficiency of `PatterNet` on various datasets and networks in terms of computation and memory reduction and compare the functionally-verified synthesized `PatterNet` accelerator with previous works.

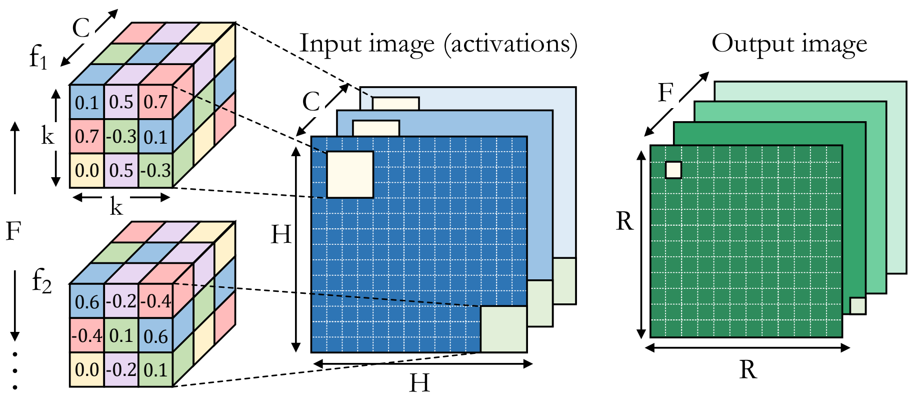


**Figure 6.2.** Convolution operation. Filters $f_1$ and $f_2$ use the same pattern with different unique weights.

## 6.2 Patterned Neural Network

### 6.2.1 Motivation

Figure 6.2 shows the parameters of a convolution layer that comprises **F** filters of $C{\times}k{\times}k$ dimension. The depth of each filter, **C**, is equal to the number of channels (feature maps) of the input activations. An output pixel (activation) of output channel $\ell$ is created by applying the filter $F_\ell$ over a particular $C{\times}k{\times}k$ window of the input. Thus, the number of output feature maps is equal to the number of filters, **F**. Multiplication of a filter and input window is essentially a dot-product by flattening them. For an input with $H{\times}H$ channels, the output image has a dimension of $R{\times}R$, for $R = \frac{H-k}{S} + 1$, where $S$ is the stride size (i.e., the sliding step of the filters).

Assuming every $n_f$ subset of a layer's filters share the same clustering pattern, the total parameter memory consists of $C{\times}k{\times}k{\times}\log\mathbf{G}$ bits to store the common index table (i.e., cluster indexes of weights instead of values), and $n_f{\times}\mathbf{G}{\times}8\text{b}$ bits to store the actual weights of $n_f$ filters assuming 8-bit weights. The total number of operations include total $C{\times}k{\times}k$ ADD (in **G** groups/clusters), accompanied with **G** MULs and ADDs for each filter to generate an output.

That being said, Figure 6.3 shows the parameter memory and operation reduction of patterned VGG-16 layers over the 8-bit quantized model for $N{\in}\{4,8,16\}$ filters sharing one pattern assuming $\mathbf{G}{=}16$ unique weights per filter. For intermediate layers, saving ranges from

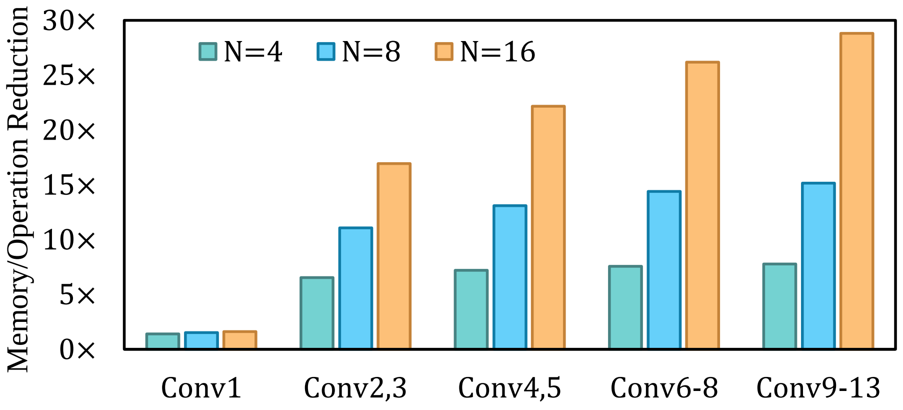


**Figure 6.3.** Memory and operation saving of patterned weight clustering when $N$ filters share one pattern.

7.2× to 22.1× depending on $N$, and up to 28.8× in the last layers. We assumed that a fixed number of filters share a single pattern. In practice, each pattern may contribute to a different number of filters of a layer as we elaborate in Section 6.2.2. Note that $\mathbf{G}=16$ and 8-bit weights are a special case that leads to the same memory and operation savings; otherwise, the savings can be different.

## 6.2.2 Pattern Selection

Pattern selection involves determining the *number* of clustering patterns, the patterns themselves, and the assignment of patterns to filters. Exploring inter-filter structural similarities is a proper starting point in determining the common patterns and the filters that share these patterns. Patterning is more complicated than other problems such as filter pruning that considers the filters exclusively (e.g., pruning based on $l_1$ norms [113] or ranks of filters [8]).

We use Figure 6.4 to elaborate our proposed pattern selection approach. Using a pre-trained model, we cluster the weights of each filter to **G** groups using any conventional approach such as $k$-means. Note that this step is a simple one-shot clustering merely to reduce the number of unique weights of filters. In this illustrative example, filters $f_1$ and $f_2$ are clustered into four groups, distinguished by different colors. In patterned clustering, for flexibility, each filter can have an arbitrary set of unique weights different than other filters (denoted by $G_{1-4}$ for $f_1$ and

$H_{1-4}$ for $f_2$ in Figure 6.4). The goal is to find the filters with most *similar* clustering, indicated by how many same-cluster weight indexes in $f_i$ are also in the same cluster in $f_j$. A naive approach is to correspond each cluster of $f_i$ with a cluster in $f_j$ and count the overlaps, which results in **G**! combinations ($2\times10^{13}$ for **G**=16).

We formulate the "similarity finding" as the **Hungarian matching** problem. For each pair of filters $f_i$ and $f_j$, we create the table of longest common subsequences between all groups, ending up in a **G**×**G** table. For instance, in Figure 6.4, cluster $G_2$ of $f_1$ has three common indexes with cluster $H_4$ of $f_2$, namely, indexes 2, 14, and 20. The Hungarian matching algorithm, with a time complexity of $O(\mathbf{G}^3)$, finds the best matching of $f_i$ and $f_j$ groups that maximizes the score (shared elements). The example of Figure 6.4 obtained a score of 20, meaning that by replacing clustering of $f_1$ with $f_2$, 20 (out of 27) weights of $f_1$ will be still in the same cluster as before (i.e., only 7 of $f_1$ weights get a different value).

We obtain the similarity scores between all pairs of filters and create an **F**×**F** distance matrix (distance defined as 1/score). Finally we use the distance matrix to find **P** (number of patterns) collections of filters, where filters of a collection have smaller distances to each other than to other collections. For this end, we use the **k-medoids** algorithm [116] to cluster **F** filters into **P** collections. Unlike *k*-means that calculates the Euclidean distance between data points, *k*-medoids works with custom cost functions, e.g., a distance matrix. In addition, unlike *k*-means, *k*-medoids returns actual data points of the collection as the center points, leading to a greater interpretability of the centers. This is essential in pattern selection as the returned centers will be the filters with their clustering pattern selected to be shared. Note that the number of filters in each of the **P** pattern collections can be different.

### 6.2.3 Free Filters

Although imposing a limited number of patterns among all the filters works for simpler datasets such as Fashion-MNIST, in more complex datasets such as CIFAR100 we observe accuracy degradation. This is a result of failing to extract certain pixel patterns because of

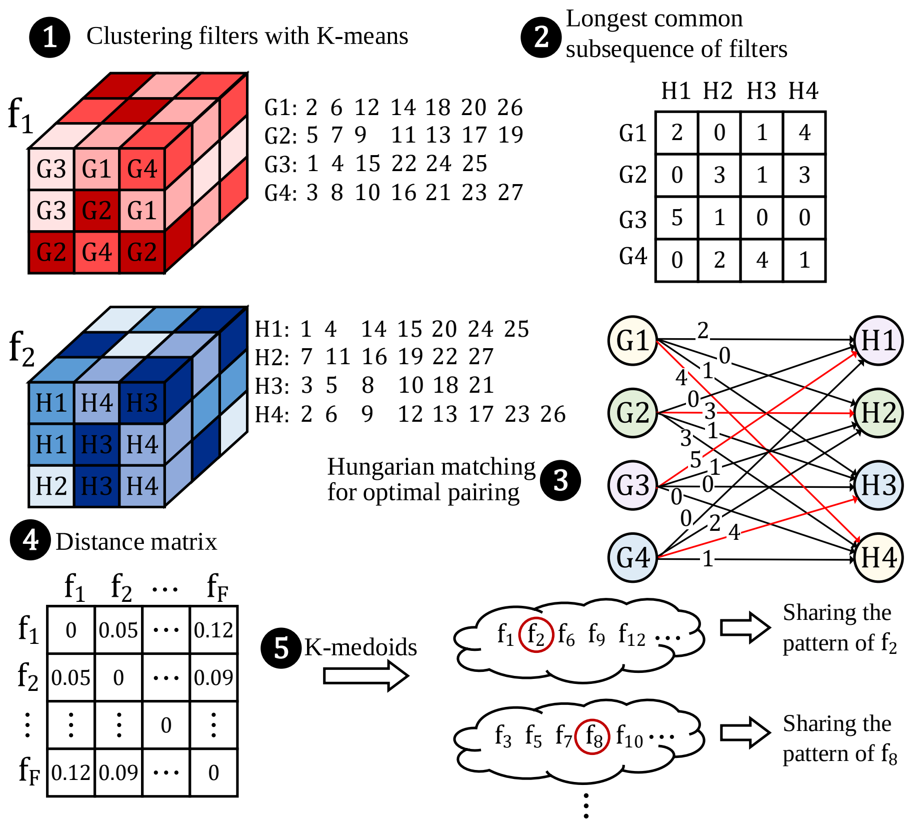


**Figure 6.4.** Selecting and sharing weight patterns.

the cluster-sharing constraint between the filters. Therefore, we relax the constraint of pattern sharing on certain filters in a layer, dubbed as **free filters**. Free filters still comply with weight clustering (hence they still benefit from factorization) but do not follow an enforced pattern.

To select the free filters, in the original pretrained model, we sort the filters based on the singular value decomposition (SVD) of their output feature maps using the train data, according to [8]. SVD value indicates how many rows of a feature map are linearly independent. The overall rank score of a filter is the mean of the generated feature maps SVDs. Filters with a rank higher than a threshold are deemed as more informative filters and selected as pattern-free (or indeed single-pattern) filters.

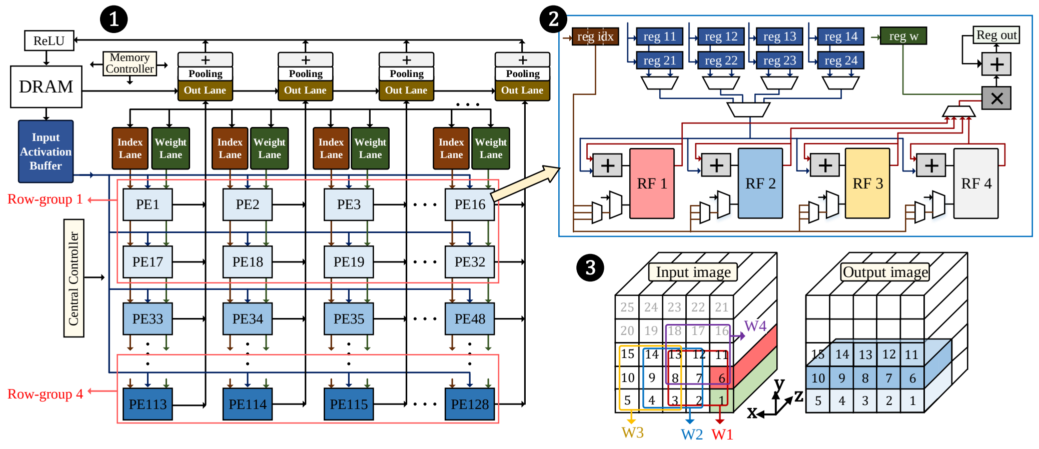


**Figure 6.5.** PatterNet (1) architecture, (2) processing element, and (3) data flow.

### 6.2.4 Patterned Model Training

After identifying the patterns associated with each filter, we use **projected gradient descent** (PGD) to calibrate the model toward the determined patterns. PGD solves constrained optimization problems, which in our case is “the solution $W$ of the DNN must belong to pattern constraints $\mathscr{Q}$”, formally, ${}_{W\in\mathscr{Q}} f(\{W^i\}_{i=1}^{\mathscr{L}}, \mathscr{X})$, where $\mathscr{L}$ is the layers and $\mathscr{X}$ is the input data. Starting from an initial $W_0 \in \mathscr{Q}$ (e.g., by cluster-wise averaging of pre-tarined weights) PGD proceeds as follows:

$$W_{k+1} = P_{\mathscr{Q}}\Big(W_k - \lambda \nabla f(W_k, \mathscr{X})\Big) \tag{6.1}$$

$P_{\mathscr{Q}}$ projects the gradients such that $W_{k+1} \in \mathscr{Q}$ as well. The projection of the gradients itself is an optimization problem:

$$P_{\mathscr{Q}}(W_k) = \arg_{W\in\mathscr{Q}} |W - W_k|_2^2. \tag{6.2}$$

Meaning that the new weights need to minimize $|W - W_k|_2^2$ while also adhering to $\mathscr{Q}$. Since weights of the solution $W$ are clustered, i.e. all weights of a cluster get the same value, the solution of (6.2) translates to minimizing $\sum(x - w_i)^2$ for each cluster, in which $w_i$s are the post-gradient weights and $x$ is the new weight of the cluster. Thus, $x = \overline{w_i}$ yields the optimal solution. Therefore, after backpropagation of each batch, we simply replace each updated weight

**Algorithm 5.** Training process in `PatterNet`

```
Inputs:  model (trained), $\mathscr{X}$, free_filters, pattern_dict, G
Output:  PatterNet model
 1: for iter from 1 to epochs×batches do
 2:    model ← SGD(model, $\mathscr{X}$)
 3:    for $\ell$ in model.convlayers do
 4:       for  in model.filters($\ell$) do
 5:          if  in free_filters[$\ell$] then
 6:             model.weight[$\ell$][] ← k-means(, G)
 7:          end if
 8:          if  in pattern_dict[$\ell$] then
 9:             model.weight[$\ell$][] ← project_weights(, pattern_dict)
10:          end if
11:       end for
12:    end for
13: end for
14: return  model =0
```

with the average of its cluster. Algorithm 5 summarizes the `PatterNet` training, where the **project_weights** function of line 9 carries out the weight projection explained above.

## 6.3 PatterNet Architecture

### 6.3.1 Overview

Figure 6.5 shows the details of `PatterNet` architecture and data flow. The architecture comprises an $R_a \times C_a$ array of processing elements (PEs). Each PE is responsible for one pattern (which is shared with one or multiple filters) and generates one/multiple output pixels. PEs gradually receive all the inputs and *pattern cluster indexes* of a window, accumulate each input in the proper group based on the index, and eventually multiply the unique weights (for all filters sharing the pattern) on the accumulated groups.

To reduce the memory accesses, `PatterNet` uses a *pattern-stationary* data flow while trying to maximize the data reuse, as well. To this end, the PE array is logically split into *row-groups*, made up of two consecutive rows (total $R_a 2$ row-groups in our architecture). All PEs in a row-group operate on the same inputs *(intra* row-group data sharing), but each PE possesses a different pattern. Thus, a row-group generates multiple channels of an output. The corresponding PEs in all row-groups (e.g., $\text{PE}_1$, $\text{PE}_{33}$, etc.) possess the same pattern (*inter* row-group data sharing), but use different inputs. Therefore, in a given time, the same channels of $R_a2$ outputs are on progress. Once all the channels associated with the running patterns are produced, `PatterNet` scans another input window to generate the next $R_a2$ outputs. After scanning all input rows, `PatterNet` starts over with the next set of patterns (if any) and repeats the same procedure to generate all the channels.

### 6.3.2 Data Flow

We elaborate the data flow of `PatterNet` using the $3\times3$ example convolution of Figure 6.53. A brick is a complete $1\times1$ window that includes all the channels ($z$ dimension). `PatterNet` fetches the input activations as *sub-bricks*. The number of channels (pixels) in a sub-brick is architectural parameter (e.g., four pixels). As shown in the figure, the convolution involving the input activation window $w1 = \left(\begin{smallmatrix} 13 & 12 & 11 \\ 8 & 7 & 6 \\ 3 & 2 & 1 \end{smallmatrix}\right)$ and the associated filter generates the right-most pixel of the output feature map. To do this, fetching of inputs starts from the bottom-right brick toward to top-left in a column-wise fashion (i.e., $1 \rightarrow 6 \rightarrow \cdots 13$) by fetching all sub-bricks commencing the next brick. This facilitates a great degree of data reuse as explained next in subsection 6.3.3. Once a sub-brick is fetched, it is broadcast to all PEs in a row-group. Along with the inputs, each PE receives the pattern index corresponding to the fetched activations.

To recap, we first create activation sub-groups by adding cluster-specific activations, before multiplying with the cluster's weight value. To implement this, in every cycle, a PE processes one activation and adds it to the corresponding cluster group (out of **G**). After fetching and accumulating all the input bricks of an input window, each PE fetches the actual weights associated with the processed pattern. For each filter that shares the current pattern, the PE fetches its **G** unique weights cycle by cycle and multiplies with the accumulated values of group-1 to group-**G**. The aforementioned window $w1$ produces the output pixels associated with 32 patterns of $\text{PE}_1$ to $\text{PE}_{32}$ of output brick 1 (i.e., at least 32 channels of the output feature map).

The convolution window is then shifted left. Hence, the row-group 1 will generate the same channels of output brick 2 as it did for output brick 1.

Multiple row-groups generate multiple output rows simultaneously. As row-group 1 processes input window $w1$, row-group 2 processes window $w4$ to generate 2nd output *row*. All row-groups generate the same channels since they use the same patterns (hence, filters). Once the row-groups finish scanning the current input rows (i.e., the windows reach the left edge), each input window moves up by $R_a 2$ (number of row-groups) rows. After scanning all the rows, `PatterNet` starts over from the first row with a new set of patterns until all output channels are created.

### 6.3.3 Data Reuse

`PatterNet` takes advantage of multiple levels of data sharing. The input activations are shared among all PEs of a row-group, and clusters index data are shared between all corresponding PEs in the row-groups (e.g., $PE_1$, $PE_{33}$, $PE_{65}$, etc.). In addition, except the edge of the image, in a $3\times3$ convolution window, an input brick is shared between three windows of the same row. E.g., in Figure 6.53, input brick 3 is used in windows $w1$, $w2$, and $w3$ (processed by `RF1`, `RF2`, and `RF3` as explained in the next subsection). Therefore, once a sub-brick of input brick 3 is fetched, `PatterNet`'s PE processes computations for all the three windows (the PE also fetches three index data in a cycle). This results in $\sim3\times$ speed-up in addition to memory access reduction. Furthermore, the $k^{\text{th}}$ row-group processes one input row ahead of its previous row-group $k-1$. `PatterNet` buffers the input to be reused later for row-group $k-1$ and avoids DRAM accesses with a small buffer. For instance, row-group 2 starts by operating on input brick 6, which will be immediately required by row-group 1 upon finishing input brick 1. Similarly, row-group 3 starts by input brick 11, which will be required by row-group 2 after processing input brick 6. This efficient data reuse is possible due to `PatterNet`'s data flow that simultaneously runs multiple vertically-adjacent windows, and processing each window in a column-wise fashion. Thus, when scanning the input image for the current patterns, each input is fetched only once from the DRAM.

### 6.3.4 Processing Units

**Processing Elements:** Figure 6.52 shows the internals of the PE. Top blue boxes are temporary registers `reg11` to `reg24` that store the activations sub-bricks fetched from the input buffer. PEs of different row-groups use the same input buffer bus in a time-multiplexed fashion. Thus, these registers are required to store enough inputs until the round-robin arbiter grants access to a row-group to fetch the next sub-brick after $R_a2$ cycles.

**Register Files:** As explained in subsection 6.3.3, an input brick may participate in several adjacent windows. The register files `RF1` to `RF4` receive one input activation as data, along with several cluster indexes as the address to accumulate the input with the proper group. One of the `RF`s is spare to avoid stalls, explained below. The `reg idx` (index register) continuously fetches these index data from the `Index Lane` buffer. Since the windows sharing an input are adjacent (i.e., an activation only differs in $x$ dimension within the windows), the index data of these windows can be aligned in one memory row. Note that since corresponding PEs of row-groups process the same pattern, the fetched index data is broadcast to all of $R_a2$ corresponding PEs of all row-groups using the common index bus of a column.

**Accumulator:** Once all inputs of a window are accumulated in an `RF`, the PE loads unique weights $w_1$ to $w_{\mathbf{G}}$ one-by-one from the `Weight Lane` to the `reg w`, and reads the accumulated sum of group-1 to group-**G** from that `RF`, accumulates the multiplications in the `reg out`, and finally transfers the output to `Out Lane`. Since each filter sharing a pattern has its own unique weights, these multiplications need to be repeated for all filters sharing the pattern. The key benefit of `PatterNet` is that, once the input sub-groups are computed a pattern, producing new output channels (of shared filters) takes just **G** cycles per filter. Since the first window (of horizontally-adjacent windows) is several input bricks ahead from the other two, in a given time, the results of only one window becomes ready in a PE. A PE contains one extra `RF`, so when an `RF` is stuck to finalize the multiplications, the fourth `RF` replaces it to process new input bricks and avoid stall.

**Output Lanes:** PEs in a column time-multiplex the same output bus to transfer the output activation to the `Out Lane`. The bus is granted in a round-robin fashion, but it does not cause performance overhead as outputs of all PEs of a column can be transferred to `Out Lane` before generating the outputs of next window. The `Out Lane` temporarily stores a few adjacent horizontal outputs (from the same PE), or adjacent vertical outputs (from the *corresponding* PEs of different row-groups) for pooling operation before writing to DRAM. The output data layout written into the DRAM is the same as input bricks, i.e., continuous pixels of an output brick are written in the same DRAM row.

## 6.4 Experiments and Results

### 6.4.1 Experimental Setup

We implemented `PatterNet` concepts (i.e., pattern and rank-based free filter selection and training) using PyTorch. For training, we used SGD optimizer, momentum of 0.9 with weight decaying, and learning rate from 0.1 down to 0.0008 over 100 epochs. For parameter **G** (number of unique weights or clusters per pattern) we found **G**=16 sufficient to retain accuracy by sweeping across a spectrum of values. Similarly, we tried a range of values for **P** (number of patterns) and found **P**=16 sufficient for accuracy.

We implemented `PatterNet` accelerator in SystemVerilog and verified its functionality with Modelsim. We synthesized it using TSMC 40 nm standard cell library at 0.9 V using Synopsys Design Compiler for a target frequency of 500 MHz. We used Artisan memory compiler with the same technology to generate SRAM buffers and register files. Power consumption of all elements is obtained using Synopsys Power Compiler. For DRAM access energy model we used Destiny [110]. Our primary architecture consists of $R_a$=8 rows (four row-groups) and $C_a$=16 (32 PEs per row-group).

### 6.4.2 Operation and Memory Reduction

We evaluate the effectiveness of `PatterNet` by comparing it with a state-of-the-art filter pruning approach dubbed Hrank [8]. We use VGG16, Resnet18, and Resnet50 networks with CIFAR10 and CIFAR100 datasets as used in [8], and a 200-class subset of ImageNet (Tiny ImageNet). To recap from Section 6.2.1, the *patterned* filters run ADDs to accumulate the input activations for **P** filters, followed by MULs of their unique weights on the resulted groups. The *free* filters are special cases of patterned filters, where a free filter has one independent pattern. Thus, free filters also benefit from factorization to reduce the number of MULs, as well as weight clustering to reduce memory.

Table 6.1 summarizes the accuracy, operation count (ADD and MUL), and memory for the aforementioned models and datasets. The *Base* column indicates the baseline 8-bit model, and *Hrank* column is the state-of-the-art filter pruning [8]. We selected the pruning ratios of Hrank layers according to its original work [8].

**CIFAR10:** As compared to the baseline VGG16 network, while HRank provides 56.1% reduction in operation count and 62.2% reduction in parameters, our method offers 72.4% reduction in operation count and 77.9% reduction in parameters, with 0.3% better accuracy. For residual networks such as ResNet18, while the operation reduction in HRank is 54.4%, our method offers 69.4% reduction. We observe a similar trend for ResNet50; 68% operation reduction in PatterNet as compared to 46% reduction of HRank. PatterNet shrinks parameters size significantly (80.2% vs HRank's 66.8%) for Resnet18 and (64.1% vs HRank's 45.7%) for Resnet50, along with better accuracy metrics as compared to HRank.

**CIFAR100:** For CIFAR100, we achieve 73.1% operation count reduction using VGG16, 61.5% using ResNet18 and 68.6% using ResNet50. The reduction in parameters is considerably better than HRank's reductions (77.4% vs 61.1%, 71% vs 48.8% and 64% vs 46.2%) for VGG16, ResNet18 and ResNet50 respectively.

**TinyImageNet:** We observe a similar trend with the Tiny ImageNet dataset. Along with an

**Table 6.1.** Comparing PatterNet with baseline and Hrank [8].

| | Model | Accuracy | | | Operation (M) | | | Parameters (MB) | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | Base | Hrank | PatterNet | Base | Hrank | PatterNet | Base | Hrank | PatterNet |
| CIFAR10 | VGG16 | 91.73% | 91.89% | 92.19% | 314.0 | 137.7 (56.1%) | 86.7 (72.4%) | 14.28 | 5.39 (62.2%) | 3.14 (77.9%) |
| | Resnet18 | 93.84% | 93.14% | 93.59% | 555.5 | 253.2 (54.4%) | 169.7 (69.4%) | 10.64 | 3.53 (66.8%) | 2.10 (80.2%) |
| | Resnet50 | 94.65% | 94.12% | 94.64% | 1298 | 701.2 (46.0%) | 416.5 (68.0%) | 22.3 | 12.16 (45.7%) | 8.04 (64.1%) |
| CIFAR100 | VGG16 | 70.45% | 69.84% | 70.15% | 312.0 | 132.4 (57.6%) | 84.1 (73.1%) | 14.26 | 5.55 (61.1%) | 3.22 (77.4%) |
| | Resnet18 | 75.30% | 74.19% | 74.83% | 555.5 | 341.1 (38.6%) | 213.8 (61.5%) | 10.69 | 5.47 (48.8%) | 3.09 (71.0%) |
| | Resnet50 | 77.21% | 76.12% | 76.92% | 1298 | 682.1 (47.4%) | 407.5 (68.6%) | 22.5 | 12.10 (46.2%) | 8.1 (64.0%) |
| Tiny-ImageNet | VGG16 | 56.95% | 53.16% | 55.90% | 1272 | 549 (56.8%) | 355 (72.0%) | 22.79 | 14.1 (38.2%) | 11.7 (48.4%) |
| | Resnet18 | 62.28% | 60.97% | 62.20% | 2221 | 1364 (38.6%) | 854 (61.5%) | 10.74 | 5.52 (48.5%) | 3.14 (70.7%) |
| | Resnet50 | 64.20% | 62.65% | 63.88% | 5192 | 2727 (47.5%) | 1629 (68.6%) | 22.75 | 12.3 (45.8%) | 8.31 (63.4%) |

**Table 6.2.** Memory size of baseline PatterNet architecture.

| Input buffer | Index lane | Weight lane | Out lane | Register File |
|---|---|---|---|---|
| 2048×32b (8 KB) | 768×24b (2.25 KB) | 64×8b (64 B) | 512×20b (1.25 KB) | 16×20b (40 B) |

improved operation reduction (up to 72%) and parameter reduction (up to 70.7%) as compared to the baseline, our improvements are better than HRank while achieving improved accuracy metrics (1-2%) over HRank.

In summary, PatterNet shrinks the model memory up to 80.2% and operation count up to 73.1%, with a similar accuracy as compared to the 8-bit baseline models.

### 6.4.3 PatterNet Accelerator Details

The baseline `PatterNet` architecture consists of four row-groups ($R_a$=8) and 16 columns ($C_a$=16). Table 6.2 reports the size of `PatterNet` memories. As explained in Section 6.3.3, the *input buffer* stores the entire brick of a row-group for reuse by the preceding row-group. The image depth goes up to 2048 channels in Resnet50, thus, the input buffer should store 2048×4 input activations of four row-groups, packed as 2048×32b (four inputs of a brick are packed in a row and fetched at once to a row-group). The *index memory* stores all 4-bit indexes, which is 512×3×3 for the largest filter. Since three indexes per pattern is read in a column (and there are

**Table 6.3.** Characterization of PatterNet components.

| Module (one) | Area ($\mu m^2$) | Leakage (mW) | Dynamic (mW) |
|---|---|---|---|
| Input Buffer | 33,284 | 0.454 | 0.563 |
| Index Lane | 15065 | 0.171 | 0.426 |
| Weight Lane | 1,744 | 0.030 | 0.034 |
| Out Lane | 10,961 | 0.118 | 0.356 |
| PE (with RFs) | 9,472 | 0.173 | 0.470 |
| Controller | 151,029 | 1.701 | 2.518 |

two patterns in a column), the memory has a $768 \times (6 \times 4)$ layout. The *weight memory* supplies the unique weights of a column's filters. Each pattern is shared with up to 32 filters, thus, it stores up to 64 weights. Similarly, the *out lane* stores all outputs generated by a column (four row-groups and 64 filters). In addition, it stores the adjacent pixels for pooling, requiring a total of 512 rows and 20-bit per row for each output pixel. Finally, each *RF* has 16 rows for accumulation of **G**=16 groups.

Table 6.3 shows the per-component area and delay of the PatterNet using the setup of subsection 6.4.1. The 8×16 architecture of PatterNet occupies an area of 1.84 mm$^2$ (at 40 nm). The compact area is mainly due to sharing a weight index lane and an output lane within an entire column, and a small input activation memory that buffers the inputs for reuse so PatterNet uses only 70 KB on-chip memory. The design consumes a peak (wost-case) power of 145.7 mW: 29.4 mW leakage, and maximum dynamic power of 116.3 mW (at 500 MHz), 34% of which is the DRAM access power. The data reuse of PatterNet makes an effective DRAM access rate of ∼1 Byte/cycle, the same rate as PEs consume inputs in a shared fashion.

### 6.4.4 PatterNet Scalability

Figure 6.6 shows the scalability of PatterNet (implementing VGG16 model) as the array size increases from 8×16 to 8×32 (2× columns), 16×16 (2× rows), and 16×32 (2× columns and rows) for 32×32 images and ImageNet-scale 224×224 images. The area in both cases is the same and input-independent. Except for the PE utilization that shows the actual

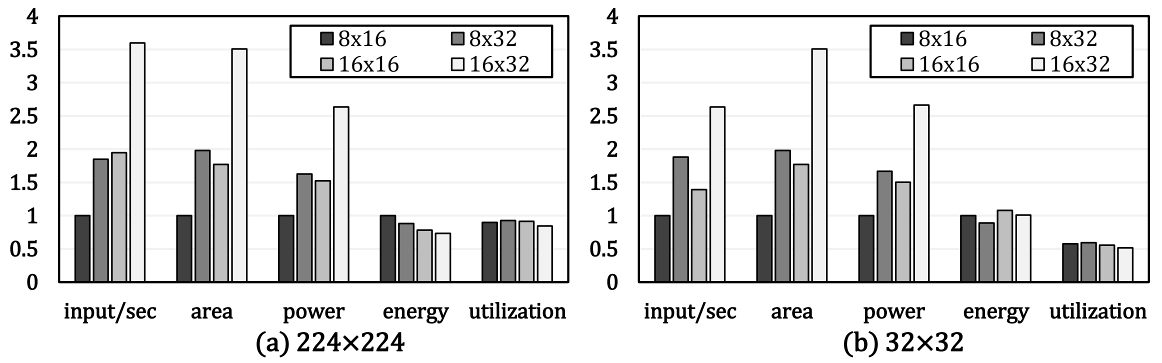


**Figure 6.6.** Scalability of performance, power, and energy of PatterNet running VGG16 with (a) 224×224 (b) 32×32 images.

**Table 6.4.** Metrics of 8×16 array for large and small images.

| **Image size** | Input/sec | Area (mm$^2$) | Power (mW) | Energy/input (mJ) |
|---|---|---|---|---|
| 224×224 | 11.3 | 1.84 | 133.6 | 11.9 |
| 32×32 | 415.3 | 1.84 | 96.3 | 0.23 |

quantities, the other parameters are normalized to 8×16 array values (Table 6.4 shows the actual values of the baseline 8×16 architecture). For large images, `PatterNet` architecture shows better scalability, i.e., 3.6× higher performance (input/sec) when both rows and columns duplicate. However, for small images, PE utilization rate reduces down to 46% in the 16×32 array. As a result, it achieves only 2.6× performance gain. The average utilization rate for large images is 90% in the baseline 8×16 array and 76% in the largest array. The area is *not* scaled by 4× since the size of index lane and weight lane buffers remains the same, and their number only increases by 2×. Finally, for large images, the largest array (16×32) shows better energy/input. This is mainly because the DRAM access power ratio significantly reduces (down to 9.3%) because the fetched inputs are reused between more row-groups.

### 6.4.5 Comparison with Previous Work

We compare performance-per-watt of `PatterNet` with the state-of-the-art FuseKNA [7] which also reuses the overlapping ADDs among kernels in a bit-serial accelerator, and with SCNN [6] which is a MAC-based sparse (zero-skipping) accelerator (results compiled from [7]).

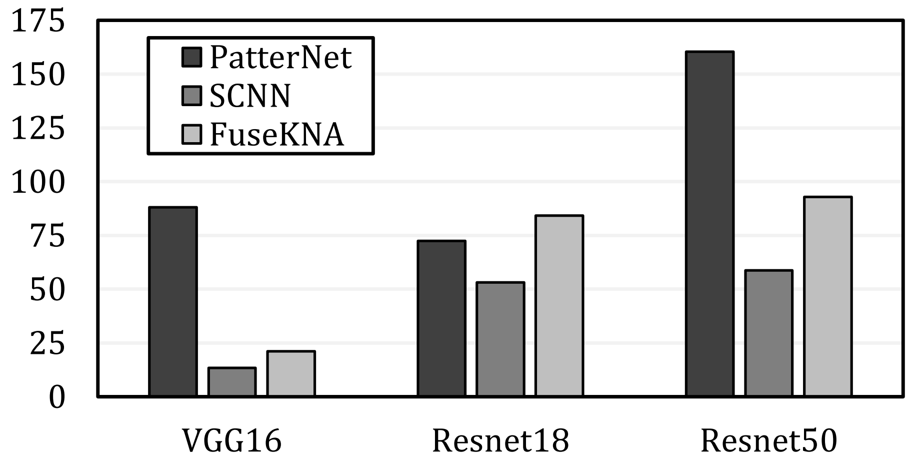


**Figure 6.7.** Comparison of PatterNet, SCNN [6], and FuseKNA [7] energy efficiency normalized to GPU.

Figure 6.7 shows the performance-per-watt (energy per image) normalized to Nvidia 1080 GTX GPU, all designs running 224×224 images. `PatterNet` surpasses GPU energy efficiency by 107×, SCNN by 3.6×, and FuseKNA by 2.2×.

### 6.4.6 Physical Design and Tapeout

We use TSMC 40nm foundry enablement to realize the silicon implementation and fabrication of `PatterNet` accelerator DIE. The DRC and LVS clean layout of `PatterNet` accelerator is shown in Figure 6.8. As shown in the figure, we incorporate am additonal capability of Hyperdimensional (HD) compute logic [117], which is a new brain-inspired computing paradigm to build lightweight learning algorithms. The traditional CNN pipeline is replaced by HDnn encoding pipeline (the last few layers replaced by HD), to accelerate the computing, improving training performance and energy efficiency. For more details of the implementation details of HD logic in `PatterNet` we refer the readers to tinyHD [118].

For the physical design, we use six-layer metal stack and a restribution layer (RDL); M1, M2, M3, M4, M5, M6, and AP. For the interface with package, we use wirebond design with 145 IO pads (8 power and ground pads) and each IO pad is enabled with an aluminium launching bondpad. Since the DIE area is limited by the number of IO pads, we perform multiplexing of

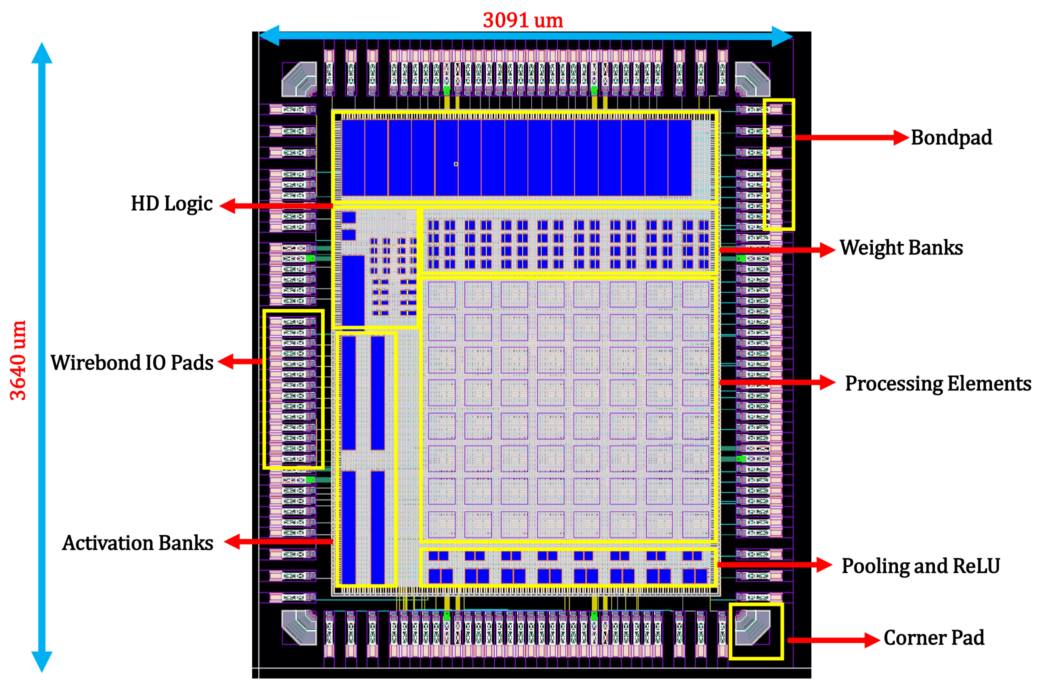


**Figure 6.8.** Physical layout of `PatterNet` accelerator; wirebond design with 145 IO pads and a total silicon area of $11.25mm^2$ in TSMC40nm.

IO signals, to maximally optimize the number of IO pads. The layout area is 3091 $\times$ 3640 um. The fabrication phase of the tapedout chip is handled by TSMC's multi-project wafer (MPW) Cybershuttle program. The subsequent design, fabrication and assembly of package and boards is outsourced to Siltronics [119].

## 6.5 Conclusion

In this work, we introduced the concept of patterned cluster sharing between DNNs filters, which achieves memory reduction by reusing the clustering indexes, and operation reduction by using weight factorization and reusing the result among the filters of the same cluster. We proposed techniques to determine and assign the patterns over the filters, as well as a training approach to yield the target patterns. We evaluated the filter patterning using different datasets and networks, which revealed its effectiveness in significant memory and operation reduction (by 80.2% and 73.1% respectively), which surpassed the state-of-the-art filter pruning technique while achieving better accuracy. We also proposed PatterNet accelerator based on the above ideas, which obtained 2.2$\times$ better energy efficiency than state-of-the-art accelerators.

### 6.5.1 Acknowledgements

PatterNet was supported by TSMC, in part by CRISP, one of six centers in JUMP (an SRC program sponsored by DARPA), SRC Global Research Collaboration (GRC) grant, and NSF grants #1911095, #1826967, #2100237, and #2112167. The substrate package and the printed circuit board design, fabrication and assembly of PatterNet is supported by Silitronics.

Chapter 6 contains materials from “PatterNet: Explore and Exploit Filter Patterns for Efficient Deep Neural Networks”, by Uday Mallappa*, Behnam Khaleghi*, Duygyu Yaldiz, Haichao Yang, Monil Shah, Jaeyoung Kang and Tajana Rosing, which appears in Design Automation Conference, July 2022. The dissertation author was one of the primary investigators and authors of this paper.

# Chapter 7

# Conclusions

## 7.1 Thesis Summary

The device scaling over the last three decades has helped the chip industry exhibit consistent performance, power, and area (PPA) gains. One of the primary contributors to the PPA gain is the EDA industry, which helped designers realize optimal multi-objective design implementations. However, as the chip industry transitions into sub-7nm nodes, more complex rules increase the overall design cost and tool runtimes. To keep the design cost and turn-around-time under control, designers introduce margins at various stages of the IC design flow, and also compromise on the design-space exploration, leading to unclaimed benefits from the newer nodes. Therefore, to keep up with the pace of PPA expectations and also fight the saturation of Moore's law, we focus on two promising opportunities at the top of the compute stack: (i) AI for Design Optimization, and (ii) ASIC Design for accelerating AI algorithms.

Chapters 2, 3, and 4 present various applications of AI for efficient IC design. Our PBA-GBA model in Chapter 2 predicts expensive PBA timing results from inexpensive GBA results, essentially addressing the accuracy-runtime tradeoff during the timing analysis. We demonstrated that our model-predicted PBA arrival times reduce mean, 99th percentile, and max divergence metrics by at least 26.6%, 13.4% and 11.7%, respectively, as compared to reference PBA-GBA divergence metrics. In the same chapter, our corner prediction framework accurately captures and exploits the physics of timing delays across multiple corners to improve design convergence and design cost. With a 1M-instance example in foundry 16nm enablement (10% training, 90% testing), our corner prediction model based on 10 observed corners predicts timing results at the remaining 48 unobserved corners with less than 0.5% relative root mean squared error, and 99th percentile relative prediction error of less than 0.6%. Chapter 3 formulates the detailed placement problem of ASIC design, as a Markov Decision Process (MDP). Our two-stage `RLPlace` framework utilizes Deep-Q learning for coarse arrangements of clusters and Satisfiability Modulo Theories (SMT) for fine-grain refinement. With the global placement output of two critical IPs as the start point, we achieve up to 1.35% improvement as compared to commercial tool's detailed-placement. Chapter 4 presents an SMT-based joint application-aware routing and static VC assignment framework that guarantees deadlock freedom, to achieve optimal latency in network-based communication subsystems. Our experiments show that our approach can achieve up to 30% better performance than the state-of-the-art application-aware oblivious routing algorithms.

In Chapter 5 and Chapter 6, we present two domain-specialized hardware accelerators, focusing on the inference phase of neural network based image classification applications. We alleviate the computational and energy burden of neural network accelerators, by reusing the computations, and skipping unnecessary computations. Our `TermiNETor` framework in Chapter 5 is an algorithmic innovation with hardware support, for dynamically predicting and skipping the ineffectual outputs. `TermiNETor` demonstrates up to 1.7× speedup with negligible loss of accuracy across many different networks and datasets. The `TermiNETor` accelerator exploits the dynamic bitwise convolution terminations with an average energy efficiency of 120× as compared to GPU, and up to 4.84× with respect to the state-of-the-art sparsity-aware accelerators. To exploit the reuse of computations and reduce the memory footprint, Chapter 6 proposes a novel patterned neural network `PatterNet`. We propose a systematic framework for exploring and enforcing shared cluster topologies to optimize computation reuse, and an efficient hardware implementation of our idea. We demonstrate a significant memory and operation reduction (by 80.2% and 73.1% respectively), surpassing the state-of-the-art filter pruning technique while achieving better accuracy. We also designed `PatterNet` accelerator that is 2.2× more energy efficient than state-of-the-art accelerators. We use TSMC 40nm foundry enablement for the silicon realization of `PatterNet` and `TermiNETor` neural network accelerators.

## 7.2 Future Directions

AI-based techniques for multidisciplinary IC design analysis and optimization, and task-optimized silicon implementation offer a viable way to extend the golden age of semiconductor innovation, despite the stalling of Moore's law and increased IC design costs for newer nodes.

### 7.2.1 AI for Design Optimization

In this dissertation, we focused on isolated tasks of the IC design process. Integrating multiple AI-based models at various stages of IC design and translating them into end-to-end PPA benefits is an important future direction to pursue. To this end, we seek to integrate our PBA-GBA and corner prediction models with an academic sizer and optimizer, to explore the benefit from reduced pessimism in multi-corner multi-mode (MCMM) timing closure and sizing for leakage and total power reduction. For the RL-based detailed-placement optimization, we use half-perimeter wire-length (HPWL) as our reward function. As part of our future work, we are exploring other rewards such as routing congestion, placement density, and timing criticality. On the other hand, the demand for big data to support AI for EDA has been increasing. However, the lack of circuit benchmarks severely hinders the research outcomes. To mitigate the lack of open source benchmarks, we are generating large synthetic circuit repositories, intended for various transfer learning problems in the IC design cycle.

Our SMT-based joint routing and virtual channel (VC) allocation framework, JARVA guarantees a deadlock free solution for the mesh topology. As part of our future work, we seek to extend the formulation to other topologies like Torus and Folded-Torus.

### 7.2.2 ASIC Design for AI Acceleration

An AI-specific accelerator is a hardware implementation specialized to meet the compute, memory, and energy requirements of resource-constrained AI implementations. In this dissertation, we propose two neural network accelerators that are based on reusing redundant computations (`PatterNet`) and terminating unnecessary computations (`TermiNETor`). Though each of these proposed accelerators offers significant performance and energy benefits, combining the reuse and early termination ideas can offer the best of both frameworks. However, designing the hardware implementation and formulating a data flow that can support both patterned neural networks and early dynamic termination is a challenging task. In addition, by exploiting some of the physical design knobs, such as power gating and clock gating for PE rows, multi-voltage islands can further optimize energy requirements.

# Bibliography

[1] H. Jones, "International Business Strategies (IBS)." https://www.ibs-inc.net.

[2] https:ai-startups.org.

[3] N. P. Jouppi, C. Young, N. Patil, and D. Patterson, "A domain-specific architecture for deep neural networks," *Commun. ACM*, vol. 61, p. 50–59, aug 2018.

[4] R. Penrose, *The Road to Reality: A Complete Guide to the Laws of the Universe*. Jonathan Cape, 2004.

[5] C. Glass and L. Ni, "The turn model for adaptive routing," in *Proceedings ISCA*, 1992.

[6] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "Scnn: An accelerator for compressed-sparse convolutional neural networks," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 2, pp. 27–40, 2017.

[7] J. Yang *et al.*, "Fusekna: Fused kernel convolution based accelerator for deep neural networks," in *International Symposium on High-Performance Computer Architecture*, pp. 894–907, 2021.

[8] M. Lin *et al.*, "Hrank: Filter pruning using high-rank feature map," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 1529–1538, 2020.

[9] "Eda vendors should improve the runtime performance of path-based timing analysis." https://www.electronicdesign.com/eda/eda-vendors-should-improve-runtime-performance-path-based-analysis, 2013.

[10] T. N. Theis and H.-S. P. Wong, "The end of moore's law: A new beginning for information technology," *Computing in Science  Engineering*, vol. 19, no. 2, pp. 41–50, 2017.

[11] C. E. Leiserson, N. C. Thompson, J. S. Emer, B. C. Kuszmaul, B. W. Lampson, D. Sanchez, and T. B. Schardl, "There's plenty of room at the top: What will drive computer performance after moore's law?," *Science*, vol. 368, no. 6495, 2020.

[12] Y. Sherry and N. C. Thompson, "How fast do algorithms improve? [point of view]," *Proceedings of the IEEE*, vol. 109, no. 11, pp. 1768–1777, 2021.

[13] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems* (F. Pereira, C. Burges, L. Bottou, and K. Weinberger, eds.), vol. 25, Curran Associates, Inc., 2012.

[14] J. Kober and J. Peters, *Reinforcement Learning in Robotics: A Survey*, pp. 9–67. Cham: Springer International Publishing, 2014.

[15] S. Mohanty, A. Vijay, and N. Gopakumar, 2022.

[16] D. Bahdanau, J. Chorowski, D. Serdyuk, P. Brakel, and Y. Bengio, "End-to-end attention-based large vocabulary speech recognition," in *2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 4945–4949, 2016.

[17] H. Fan, T. Murrell, H. Wang, K. V. Alwala, Y. Li, Y. Li, B. Xiong, N. Ravi, M. Li, H. Yang, J. Malik, R. Girshick, M. Feiszli, A. Adcock, W.-Y. Lo, and C. Feichtenhofer, "PyTorchVideo: A deep learning library for video understanding," in *Proceedings of the 29th ACM International Conference on Multimedia*, 2021. https://pytorchvideo.org/.

[18] Z. Chen, Z. Peng, X. Zou, and H. Sun, "Deep learning based anomaly detection for muti-dimensional time series: A survey," in *Cyber Security* (W. Lu, Y. Zhang, W. Wen, H. Yan, and C. Li, eds.), (Singapore), pp. 71–92, Springer Nature Singapore, 2022.

[19] Y. Lin and D. Z. Pan, *Machine Learning in Physical Verification, Mask Synthesis, and Physical Design*, pp. 95–115. Cham: Springer International Publishing, 2019.

[20] G. Huang, J. Hu, Y. He, J. Liu, M. Ma, Z. Shen, J. Wu, Y. Xu, H. Zhang, K. Zhong, X. Ning, Y. Ma, H. Yang, B. Yu, H. Yang, and Y. Wang, "Machine learning for electronic design automation: A survey," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 26, jun 2021.

[21] S. Dai, Y. Zhou, H. Zhang, E. Ustun, E. F. Young, and Z. Zhang, "Fast and accurate estimation of quality of results in high-level synthesis with machine learning," in *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 129–132, 2018.

[22] I. Wagner, V. Bertacco, and T. Austin, "Microprocessor verification via feedback-adjusted markov models," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 6, pp. 1126–1138, 2007.

[23] O. Guzey, L.-C. Wang, J. Levitt, and H. Foster, "Functional test selection based on unsupervised support vector analysis," in *2008 45th ACM/IEEE Design Automation Conference*, pp. 262–267, 2008.

[24] D. Hyun, Y. Fan, and Y. Shin, "Accurate wirelength prediction for placement-aware synthesis through machine learning," in *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 324–327, 2019.

[25] W.-K. Cheng, Y.-Y. Guo, and C.-S. Wu, "Evaluation of routability-driven macro placement with machine-learning technique," in *2018 7th International Symposium on Next Generation Electronics (ISNE)*, pp. 1–3, 2018.

[26] E. C. Barboza, N. Shukla, Y. Chen, and J. Hu, "Machine learning-based pre-routing timing prediction with reduced pessimism," in *2019 56th ACM/IEEE Design Automation Conference (DAC)*, pp. 1–6, 2019.

[27] V. A. Chhabria, A. B. Kahng, M. Kim, U. Mallappa, S. S. Sapatnekar, and B. Xu, "Template-based pdn synthesis in floorplan and placement using classifier and cnn techniques," in *2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 44–49, 2020.

[28] Z. Xie, H. Ren, B. Khailany, Y. Sheng, S. Santosh, J. Hu, and Y. Chen, "Powernet: Transferable dynamic ir drop estimation via maximum convolutional neural network," *2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 13–18, 2020.

[29] U. Mallappa and C.-K. Cheng, "Gra-lpo: Graph convolution based leakage power optimization," in *2021 26th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 697–702, 2021.

[30] Y.-C. Lu, S. Nath, S. Pentapati, and S. K. Lim, "Eco-gnn: Signoff power prediction using graph neural networks with subgraph approximation," *ACM Trans. Des. Autom. Electron. Syst.*, 2022.

[31] S. Nath, G. Pradipta, C. Hu, T. Yang, B. Khailany, and H. Ren, "Generative self-supervised learning for gate sizing: Invited," DAC '22, p. 1331–1334, 2022.

[32] R. Chen, W. Zhong, H. Yang, H. Geng, X. Zeng, and B. Yu, "Faster region-based hotspot detection," in *2019 56th ACM/IEEE Design Automation Conference (DAC)*, pp. 1–6, 2019.

[33] D. Silver, A. Huang, C. Maddison, A. Guez, L. Sifre, G. Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, pp. 484–489, 01 2016.

[34] A. Mirhoseini, A. Goldie, M. Yazgan, J. Jiang, E. M. Songhori, S. Wang, Y.-J. Lee, E. Johnson, O. Pathak, S. Bae, A. Nazi, J. Pak, A. Tong, K. Srinivasa, W. Hang, E. Tuncer, A. Babu, Q. V. Le, J. Laudon, R. Ho, R. Carpenter, and J. Dean, "Chip placement with deep reinforcement learning," *ArXiv*, vol. abs/2004.10746, 2020.

[35] T. Hamada, C.-K. Cheng, and P. Chau, "An efficient multi-level placement technique using hierarchical partitioning," in *1991., IEEE International Sympoisum on Circuits and Systems*, pp. 2044–2047 vol.4, 1991.

[36] J. Z. Yan, N. Viswanathan, and C. Chu, "Handling complexities in modern large-scale mixed-size placement," in *2009 46th ACM/IEEE Design Automation Conference*, pp. 436–441, 2009.

[37] Y.-H. Huang, Z. Xie, G.-Q. Fang, T.-C. Yu, H. Ren, S.-Y. Fang, Y. Chen, and J. Hu, "Routability-driven macro placement with embedded cnn-based prediction model," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 180–185, IEEE, 2019.

[38] N. Froleyks, M. Heule, M. Iser, M. Järvisalo, and M. Suda, "Sat competition 2020," *Artificial Intelligence*, vol. 301, p. 103572, 2021.

[39] Bjørner *et al.*, "$\nu$z - an optimizing smt solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, 2015.

[40] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *arXiv preprint arXiv:1512.03385*, 2015.

[41] C. Manning, M. Surdeanu, J. Bauer, J. Finkel, S. Bethard, and D. McClosky, "The stanford corenlp natural language processing toolkit," 01 2014.

[42] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," ASPLOS '14, p. 269–284, 2014.

[43] Y.-H. Chen, T.-J. Yang, J. Emer, and V. Sze, "Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 9, no. 2, pp. 292–308, 2019.

[44] "Synopsys primetime user guide." https://www.synopsys.com/Tools/\Implementation/SignOff/Pages/PrimeTime.aspx.

[45] "Cadence tempus user guide." https://www.cadence.com/content/cadence-www/global/en_US/home/tools/digital-design-and-signoff/silicon-signoff/tempus-timing-signoff-solution.html.

[46] "Opencores." https://opencores.org.

[47] "Risc-v." https://riscv.org.

[48] M. M. Ozdal, C. Amin, A. Ayupov, S. Burns, G. Wilke, and C. Zhuo, "The ispd-2012 discrete cell sizing contest and benchmark suite," in *Proceedings of the 2012 ACM International Symposium on International Symposium on Physical Design*, ISPD '12, p. 161–164, 2012.

[49] A. B. Kahng, "Machine learning applications in physical design: Recent results and directions," in *Proceedings of the 2018 International Symposium on Physical Design*, ISPD '18, (New York, NY, USA), p. 68–73, Association for Computing Machinery, 2018.

[50] "Tau workshop." https://www.tauworkshop.com.

[51] J. Bhasker and R. Chadha, *Static Timing Analysis for Nanometer Designs: A Practical Approach*. Springer, 2009.

[52] J.-J. Nian, S.-H. Tsai, and C.-Y. Huang, "A unified multi-corner multi-mode static timing analysis engine," in *2010 15th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 669–674, 2010.

[53] S.-S. Han, A. B. Kahng, S. Nath, and A. S. Vydyanathan, "A deep learning methodology to proliferate golden signoff timing," in *2014 Design, Automation  Test in Europe Conference Exhibition (DATE)*, pp. 1–6, 2014.

[54] A. B. Kahng, M. Luo, and S. Nath, "Si for free: machine learning of interconnect coupling delay and transition effects," in *2015 ACM/IEEE International Workshop on System Level Interconnect Prediction (SLIP)*, pp. 1–8, 2015.

[55] A. B. Kahng, S. Kang, H. Lee, S. Nath, and J. Wadhwani, "Learning-based approximation of interconnect delay and slew in signoff timing tools," in *2013 ACM/IEEE International Workshop on System Level Interconnect Prediction (SLIP)*, pp. 1–8, 2013.

[56] T.-w. Huang and M. D. F. Wong, "On fast timing closure: speeding up incremental path-based timing analysis with mapreduce," in *2015 ACM/IEEE International Workshop on System Level Interconnect Prediction (SLIP)*, pp. 1–6, 2015.

[57] A. B. Kahng, M. Luo, and S. Nath, "Si for free: machine learning of interconnect coupling delay and transition effects," in *2015 ACM/IEEE International Workshop on System Level Interconnect Prediction (SLIP)*, pp. 1–8, 2015.

[58] D. Stamoulis, D. Rodopoulos, B. H. Meyer, D. Soudris, and Z. Zilic, "Linear regression techniques for efficient analysis of transistor variability," in *2014 21st IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, pp. 267–270, 2014.

[59] S. Bian, M. Hiromoto, M. Shintani, and T. Sato, "Lsta: Learning-based static timing analysis for high-dimensional correlated on-chip variations," in *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1–6, 2017.

[60] S. Onaissi, F. Taraporevala, J. Liu, and F. Najm, "A fast approach for static timing analysis covering all pvt corners," in *2011 48th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 777–782, 2011.

[61] L. G. e Silva, L. M. Silveira, and J. R. Phillips, "Efficient computation of the worst-delay corner," in *2007 Design, Automation  Test in Europe Conference  Exhibition*, pp. 1–6, 2007.

[62] S. Onaissi and F. N. Najm, "A linear-time approach for static timing analysis covering all process corners," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 7, pp. 1291–1304, 2008.

[63] C. Visweswariah, K. Ravindran, K. Kalafala, S. Walker, S. Narayan, D. Beece, J. Piaget, N. Venkateswaran, and J. Hemmett, "First-order incremental block-based statistical timing analysis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 10, pp. 2170–2180, 2006.

[64] L. Breiman, J. Friedman, C. Stone, and R. Olshen, *Classification and Regression Trees*. Taylor & Francis, 1984.

[65] J. H. Friedman, "Multivariate Adaptive Regression Splines," *The Annals of Statistics*, vol. 19, no. 1, pp. 1 – 67, 1991.

[66] T. Hastie, R. Tibshirani, and J. Friedman, *The elements of statistical learning: data mining, inference and prediction*. Springer, 2009.

[67] I. Guyon and A. Elisseeff, "An introduction to variable and feature selection," *J. Mach. Learn. Res.*, vol. 3, p. 1157–1182, mar 2003.

[68] I. Jolliffe, *Principal Component Analysis*. Springer, 2002.

[69] S. J. Russell and P. Norvig, *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited, 2016.

[70] D. Z. Pan and Y. Lin, "Machine learning and its applications in ic physical design," in *IEEE Conference on Electrical Performance of Electronic Packaging and Systems*, 2016.

[71] R. Kirby, S. Godil, R. Roy, and B. Catanzaro, "Congestionnet: Routing congestion prediction using deep graph neural networks," in *2019 IFIP/IEEE 27th International Conference on Very Large Scale Integration (VLSI-SoC)*, pp. 217–222, 2019.

[72] Y.-Y. Huang, C.-T. Lin, W.-L. Liang, and H.-M. Chen, "Learning based placement refinement to reduce drc short violations," in *2021 International Symposium on VLSI Design, Automation and Test (VLSI-DAT)*, pp. 1–4, 2021.

[73] T. Yang, G. He, and P. Cao, "Pre-routing path delay estimation based on transformer and residual framework," in *2022 27th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 184–189, 2022.

[74] L.-W. Chen, Y.-N. Sui, T.-C. Lee, Y.-L. Li, M. C.-T. Chao, I.-C. Tsai, T.-W. Kung, E.-C. Liu, and Y.-C. Chang, "Path-based pre-routing timing prediction for modern very large-scale integration designs," in *2022 23rd International Symposium on Quality Electronic Design (ISQED)*, pp. 1–6, 2022.

[75] F. Agostinelli, S. McAleer, A. Shmakov, and P. Baldi, "Solving the rubik's cube with deep reinforcement learning and search," *Nature Machine Intelligence*, vol. 1, 08 2019.

[76] A. Agnesina, K. Chang, and S. K. Lim, "Vlsi placement parameter optimization using deep reinforcement learning," in *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pp. 1–9, 2020.

[77] Y.-H. Huang, Z. Xie, G.-Q. Fang, T.-C. Yu, H. Ren, S.-Y. Fang, Y. Chen, and J. Hu, "Routability-driven macro placement with embedded cnn-based prediction model," in *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 180–185, 2019.

[78] H. Wang, K. Wang, J. Yang, L. Shen, N. Sun, H.-S. Lee, and S. Han, "Gcn-rl circuit designer: Transferable transistor sizing with graph neural networks and reinforcement learning," in *Proceedings of the 57th ACM/EDAC/IEEE Design Automation Conference*, DAC '20, IEEE Press, 2020.

[79] D. Vashisht, H. Rampal, H. Liao, Y. Lu, D. Shanbhag, E. Fallon, and L. Kara, "Placement in integrated circuits using cyclic reinforcement learning and simulated annealing," 11 2020.

[80] Y. Lin, W. Li, J. Gu, H. Ren, B. Khailany, and D. Z. Pan, "Abcdplace: Accelerated batch-based concurrent detailed placement on multithreaded cpus and gpus," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 12, pp. 5083–5096, 2020.

[81] M. A. Kinsy *et al.*, "Optimal and heuristic application-aware oblivious routing," *IEEE Transactions on Computers*, 2013.

[82] O. Lysne *et al.*, "Layered routing in irregular networks," *IEEE Transactions on Parallel and Distributed Systems*, 2006.

[83] H. Sullivan, T. R. Bashkow, and D. Klappholz, "A large scale, homogenous, fully distributed parallel machine, ii," 1977.

[84] Dally and Seitz, "Deadlock-free message routing in multiprocessor interconnection networks," *IEEE Transactions on Computers*, 1987.

[85] T. Nesson and J. S. Lennart, "Romm routing on mesh and torus networks," in *Harvard Computer Science Group Technical Report*, 1995.

[86] L. G. Valiant and G. J. Brebner, "Universal schemes for parallel communication," Association for Computing Machinery, 1981.

[87] D. Seo *et al.*, "Near-optimal worst-case throughput routing for two-dimensional mesh networks," in *Proceedings ISCA*, 2005.

[88] L.-S. Peh and W. Dally, "A delay model and speculative architecture for pipelined routers," in *Proceedings HPCA*, 2001.

[89] Y. Song *et al.*, "Improving memory efficiency in heterogeneous mpsocs through row-buffer locality-aware forwarding," *ACM TACO*, 2020.

[90] V. Catania *et al.*, "Noxim: An open, extensible and cycle-accurate network on chip simulator," in *Proceedings ASAP*, pp. 162–163, 2015.

[91] A. Ren, T. Zhang, *et al.*, "Admm-nn: An algorithm-hardware co-design framework of dnns using alternating direction methods of multipliers," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 925–938, 2019.

[92] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv preprint arXiv:1510.00149*, 2015.

[93] M. Rastegari *et al.*, "Xnor-net: Imagenet classification using binary convolutional neural networks," in *European conference on computer vision*, pp. 525–542, Springer, 2016.

[94] C. Zhu, S. Han, H. Mao, and W. J. Dally, "Trained ternary quantization," *arXiv preprint arXiv:1612.01064*, 2016.

[95] A. Zhou, A. Yao, Y. Guo, L. Xu, and Y. Chen, "Incremental network quantization: Towards lossless cnns with low-precision weights," *arXiv preprint arXiv:1702.03044*, 2017.

[96] S.-E. Chang, Y. Li, *et al.*, "Mix and match: A novel fpga-centric deep neural network quantization framework," in *International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 208–220, IEEE, 2021.

[97] Z. Song *et al.*, "Drq: dynamic region-based quantization for deep neural network acceleration," in *ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pp. 1010–1021, 2020.

[98] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen, "Cambricon-x: An accelerator for sparse neural networks," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–12, IEEE, 2016.

[99] M. Mahmoud *et al.*, "Tensordash: Exploiting sparsity to accelerate deep neural network training," in *53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 781–795, 2020.

[100] J. Albericio *et al.*, "Bit-pragmatic deep neural network computing," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 382–394, 2017.

[101] Y. Lin, C. Sakr, Y. Kim, and N. Shanbhag, "Predictivenet: An energy-efficient convolutional neural network via zero prediction," in *IEEE international symposium on circuits and systems (ISCAS)*, pp. 1–4, 2017.

[102] M. Song *et al.*, "Prediction based execution on deep neural networks," in *ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pp. 752–763, 2018.

[103] S. Cao *et al.*, "Seernet: Predicting convolutional neural network feature-map sparsity through low-bit quantization," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 11216–11225, 2019.

[104] D. Pinto, J.-M. Arnau, and A. González, “Mixture-of-rookies: Saving dnn computations by predicting relu outputs,” *arXiv preprint arXiv:2202.04990*, 2022.

[105] V. Akhlaghi *et al.*, “Snapea: Predictive early activation for reducing computation in deep convolutional neural networks,” in *ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pp. 662–673, 2018.

[106] H. Sharma *et al.*, “Bit fusion: Bit-level dynamically composable architecture for accelerating deep neural network,” in *ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pp. 764–775, 2018.

[107] P. Ramachandran, B. Zoph, and Q. V. Le, “Searching for activation functions,” *arXiv preprint arXiv:1710.05941*, 2017.

[108] D. Yu, H. Wang, P. Chen, and Z. Wei, “Mixed pooling for convolutional neural networks,” in *International conference on rough sets and knowledge technology*, pp. 364–375, Springer, 2014.

[109] P. Judd *et al.*, “Stripes: Bit-serial deep neural network computing,” in *49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–12, 2016.

[110] M. Poremba, S. Mittal, D. Li, J. S. Vetter, and Y. Xie, “Destiny: A tool for modeling emerging 3d nvm and edram caches,” in *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1543–1546, IEEE, 2015.

[111] A. Canziani, A. Paszke, and E. Culurciello, “An analysis of deep neural network models for practical applications,” *arXiv preprint arXiv:1605.07678*, 2016.

[112] S. Ye, T. Zhang, *et al.*, “A unified framework of dnn weight pruning and weight clustering/quantization using admm,” *arXiv preprint arXiv:1811.01907*, 2018.

[113] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf, “Pruning filters for efficient convnets,” *arXiv preprint arXiv:1608.08710*, 2016.

[114] Y. He, X. Zhang, and J. Sun, “Channel pruning for accelerating very deep neural networks,” in *Proceedings of the IEEE international conference on computer vision*, pp. 1389–1397, 2017.

[115] K. Hegde *et al.*, “Ucnn: Exploiting computational reuse in deep neural networks via weight repetition,” in *International Symposium on Computer Architecture (ISCA)*, pp. 674–687, 2018.

[116] L. Kaufman and P. J. Rousseeuw, “Partitioning around medoids (program pam),” *Finding groups in data: an introduction to cluster analysis*, vol. 344, pp. 68–125, 1990.

[117] P. Kanerva, “Hyperdimensional computing: An introduction to computing in distributed representation with high-dimensional random vectors,” p. 139–159, Cognitive Computation, 2009.

[118] B. Khaleghi, H. Xu, J. Morris, and T.  Rosing, “tiny-hd: Ultra-efficient hyperdimensional computing engine for iot applications,” in *2021 Design, Automation  Test in Europe Conference  Exhibition (DATE)*, pp. 408–413, 2021.

[119] https://www.silitronics.com.