UC San Diego
UC San Diego Electronic Theses and Dissertations

Title
Architecting Machines for Green Intelligence

Permalink
https://escholarship.org/uc/item/985898jd

Author
Ghodrati, Soroush

Publication Date
2023

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA SAN DIEGO

Architecting Machines for Green Intelligence

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

in

Computer Science

by

Soroush Ghodrati

Committee in charge:

Professor Hadi Esmaeilzadeh, Chair
Professor Farinaz Koushanfar
Professor Jose Martinez
Professor Steven Swanson
Professor Dean Tullsen

2023

Copyright

Soroush Ghodrati, 2023

All rights reserved.

The Dissertation of Soroush Ghodrati is approved, and it is acceptable in quality and form for publication on microfilm and electronically.

University of California San Diego

2023

DEDICATION

To my family - Mahmoud, Mitra, Sara, Sorour, and especially to my dearest, Mojan.

EPIGRAPH

“Research is the foundation for progress,
and the bedrock upon which
our future is built.”

James H. Fowler

TABLE OF CONTENTS

Dissertation Approval Page . . . iii

Dedication . . . iv

Epigraph . . . v

Table of Contents . . . vi

List of Figures . . . ix

List of Tables . . . xiv

Acknowledgements . . . xv

Vita . . . xviii

Abstract of the Dissertation . . . xx

Chapter 1 Introduction . . . 1
1.1 Motivation . . . 1
1.2 Contributions . . . 2
1.3 Acknowledgements . . . 7

Chapter 2 Ultra-Energy Efficient DNN Acceleration via Mixed-Signal Computing . . . 8
2.1 Introduction . . . 8
2.2 Wide, Interleaved, and Bit-Partitioned Arithmetic . . . 11
2.3 Switched-Capacitor Circuit Design for Interleaved Bit-Partitioning . . . 14
2.3.1 Mixed-Signal Bit-Partitioned MACC Array . . . 15
2.3.2 Low-Bitwidth Switched-Capacitor MACC . . . 16
2.4 Mixed-Signal Architecture Design for Spatial Bit-Partitioning . . . 19
2.4.1 Mixed-Signal Wide Aggregator . . . 19
2.4.2 MS-WAGG Design Decisions and Tradeoffs . . . 21
2.4.3 Hierarchically Clustered Architecture . . . 24
2.4.4 BIHIWE Instruction Set . . . 26
2.5 BIHIWE Compiler Stack . . . 27
2.6 Mitigating Analog Non-Idealities . . . 28
2.7 Evaluation . . . 32
2.7.1 Methodology . . . 32
2.7.2 Experimental Results . . . 34
2.8 Related Work . . . 41
2.9 Conclusion . . . 42
2.10 Acknowledgement . . . 42

Chapter 3 Interweaving Data-Level and Bit-Level Parallelism for Energy-Efficient Digital Acceleration 44
3.1 Introduction 44
3.2 Bit-Parallel Vector Composability 46
3.3 Architecture Design for Bit-Parallel Vector Composability 49
3.3.1 Composable Vector Unit (CVU) 50
3.3.2 Design Space Exploration and Tradeoffs 52
3.3.3 Overall Architecture 54
3.4 Evaluation 55
3.4.1 Methodology 55
3.4.2 Experimental Results 58
3.5 Related Work 61
3.6 Conclusion 62
3.7 Acknowledgement 62

Chapter 4 Balancing Specialization and Programmability for Efficient End-to-End Acceleration of Deep Neural Networks 63
4.1 Introduction 63
4.2 Diving Deeper into non-GEMM Operations 66
4.2.1 Characteristics of Non-GEMM Operations in Modern DNNs 66
4.2.2 Requirements for Non-GEMM Execution 69
4.2.3 Spectrum of Approaches to Support Non-GEMM Layers 70
4.3 Design Considerations for Tandem Processor 73
4.3.1 Memory Subsystem Design 73
4.3.2 Specialized On-Chip Data Access Mechanism 73
4.3.3 Specialized Loop Execution 75
4.3.4 Arithmetic Logic Units Design 76
4.3.5 Tandem Processor Integration with GEMM Unit 77
4.4 Microarchitecture and ISA for Tandem Processor 79
4.4.1 Tandem Processor Pipeline Microarchitecture 79
4.4.2 Synchronization Logic and Overall Execution Flow 81
4.4.3 Tandem Processor ISA 83
4.5 Compiler Support for Tandem Processor 86
4.5.1 Optimizations 86
4.5.2 Compilation Workflow 86
4.6 Evaluation 88
4.6.1 Methodology 88
4.6.2 Experimental Results 90
4.7 Related Work 97
4.8 Conclusion 98
4.9 Acknowledgement 98

Chapter 5 Cost-Effective Accelerator Utilization via Spatial Multi-Tenancy 99
5.1 Introduction 99

5.2 Dynamic Architecture Fission: Concepts and Overview 102
5.3 Architecture Design for Fission:
Challenges and Opportunities 104
5.3.1 Fission for Compute and the Need for New Communication Patterns . . . 105
5.3.2 Fission for the On-Chip Memory and the Need for Reorganizing the Entire Design 108
5.3.3 Fission without Reorganization Defeats the Purpose 110
5.4 Microarchitecture for Fission 111
5.4.1 Omni-Directional Systolic Array Design 111
5.4.2 Reorganizing the Accelerator Microarchitecture through Fission Pod Design 113
5.4.3 Planaria Overall Architecture 114
5.5 Spatial Task Scheduling 119
5.6 Evaluation 122
5.6.1 Methodology 122
5.6.2 Experimental Results 125
5.7 Related Work 134
5.8 Conclusion 136
5.9 Acknowledgement 136

Chapter 6 Leveraging Learning Algorithms to Maximize Execution Efficiency of Transformer Models 137
6.1 Introduction 137
6.2 Background and Motivation 140
6.2.1 Self-Attention Mechanism 140
6.2.2 Gradient-Based Optimization and Regularization 141
6.2.3 Motivation 142
6.3 Algorithmic Optimizations for Sparse Attention 142
6.3.1 Learned Per-Layer Pruning 143
6.3.2 Bit-Level Early-Compute Termination 146
6.4 LeOPArd Hardware architecture 148
6.4.1 Overall Architecture 149
6.4.2 Online Pruning Hardware Realization via Bit-serial Execution 151
6.4.3 Back-End Value Processing 154
6.5 Evaluation 155
6.5.1 Methodology 155
6.5.2 Accuracy and Algorithmic Optimization 159
6.5.3 Accelerator Performance Results 161
6.5.4 Architecture Design Space Exploration 166
6.6 Related Work 169
6.7 Conclusion 171
6.8 Acknowledgement 171

Bibliography 172

LIST OF FIGURES

Figure 1.1. The trends in increasing the energy consumption and cloud compute cost of training emerging neural models. 2

Figure 1.2. Theoretical peak performance utilization for Google TPUv1 [128]. 2

Figure 2.1. Wide, interleaved, and bit-partitioned mathematical formulation. 12

Figure 2.2. $\mathcal{MS}$-BPMACC and its operational modes. 14

Figure 2.3. Low-bitwidth switched-capacitor MACC. 16

Figure 2.4. Charge-domain MACC; phase by phase. 17

Figure 2.5. Basic mixed-signal dot-product engine. 20

Figure 2.6. Our mixed-signal dot-product engine ($\mathcal{MS}$-WAGG). 20

Figure 2.7. Step-by-step analysis of improvement in (a) power and (b) area. 23

Figure 2.8. Hierarchical clustered architecture 24

Figure 2.9. BIHIWE compilation stack. 27

Figure 2.10. ResNet-50 and VGG-16 accuracy after fine-tuning. 31

Figure 2.11. Iso-Power/Iso-Area speedup and energy reduction over TETRIS. 34

Figure 2.12. Energy breakdown of BIHIWE and TETRIS. 35

Figure 2.13. Comparison with other accelerators. 36

Figure 2.14. Performance comparison to GPUs. 37

Figure 2.15. Design space exploration for bit-partitioning. 38

Figure 2.16. Design space exploration for # core per cluster. 38

Figure 2.17. Design space exploration for $\mathcal{MS}$-BPMACC. 39

Figure 3.1. The landscape of DNN accelerators and how this work fits in the picture. . 45

Figure 3.2. (a) Fixed-bitwidth bit-parallel vector composability with 2-bit slicing and (b) Bit-Flexible vector composability for 4-b inputs and 2-b weights; 2x improvement in performance compared to fixed 4-bit dot-product. 49

Figure 3.3. Composable Vector Unit. 51

Figure 3.4. Design space exploration for size of bit-slicing and vector lengths for vector composability. 52

Figure 3.5. Comparison to baseline; DDR4 memory and without bitwidth heterogeneity. 57

Figure 3.6. Comparison to baseline; HBM2 memory and without bitwidth heterogeneity. 57

Figure 3.7. Comparison to BitFusion; DDR4 memory and with bitwidth heterogeneity. 58

Figure 3.8. Comparison to BitFusion; HBM2 memory and with bitwidth heterogeneity. 59

Figure 3.9. Performance-Per-Watt comparison to RTX 2080 TI GPU. 60

Figure 4.1. Union set of neural operators/layers in representative DNNs over the years. 64

Figure 4.2. Cumulative number of GEMM and non-GEMM operations across benchmarks. Last bar covers the frequency of usage across all the models. 67

Figure 4.3. Repeated subgraphs of (a) ResNet-50 [110], (b) MobileNetv2 [212], and BERT [78]. The gray ovels illustrate the non-GEMM operations and white rectangles show the GEMM-based operations. 68

Figure 4.4. Roofline model for a number of prevalent non-GEMM operators. 69

Figure 4.5. Overhead of address calculation using arithmetic instructions. "N-G" and "E2E" denote the runtime for Non-GEMM and End-to-End execution. This experiment was performed on Tandem Processor. 74

Figure 4.6. Runtime overhead of loop execution using branch logic across benchmarks. "N-G" and "E2E" denote the runtime for Non-GEMM and End-to-End execution. 75

Figure 4.7. Compute resource utilization for GEMM unit and Tandem Processor for layer level and tile level granularity of coordination. 78

Figure 4.8. The Tandem Processor pipeline microarchitecture. 79

Figure 4.9. The execution controller. 82

Figure 4.10. Tandem Processor instruction set formats that does not use any registers. . 84

Figure 4.11. Compilation workflow. 87

Figure 4.12. Accuracy of cycle-accurate simulator (normalized to RTL simulation). . . . 88

Figure 4.13. Performance comparison to offchip CPU fallback and dedicated units. . . . 92

Figure 4.14. Energy comparison to offchip CPU fallback and dedicated units. 92

Figure 4.15. Runtime breakdown in NPU-Tandem compared to baseline (1) (Offchip CPU + GEMM Unit) and baseline (2) (Dedicated Units + Offchip CPU + GEMM Unit). 92

Figure 4.16. Comparison with Gemmini [95]. 93

Figure 4.17. Speedup / multi-core RISC-V. 93

Figure 4.18. Speedup / Intel CPU integration. 95

Figure 4.19. Energy / Intel CPU integration. 95

Figure 4.20. Comparison to Jetson Xavier. 95

Figure 4.21. Fraction of non-GEMM operations to total runtime across various sizes of Tandem Processor. 95

Figure 4.22. Performance comparison to offchip CPU fallback and dedicated units for batch-size=32. 96

Figure 4.23. (a) Tandem Processor layout and (b) area breakdown. 97

Figure 5.1. Illustration of possible fission schemes of Planaria with their corresponding spatially mapped DNNs. 102

Figure 5.2. A monolithic systolic array accelerator. 104

Figure 5.3. Illustration of possible fission scenarios. 106

Figure 5.4. Omni-directional systolic execution. 107

Figure 5.5. On-chip memory fission and connection to subarrays. 109

Figure 5.6. Underutilization of the subarrays while they are connected to on-chip memory similar to conventional systolic arrays without reorganization of the design. The teal-colored subarray is the only one that can be utilized. . 110

Figure 5.7. On-chip memory to subarrays connectivity through high-radix crossbars in an alternative hypothetical design point. 111

Figure 5.8. Switching network for omni-directional systolic array. 112

Figure 5.9. Fission Pod 115

Figure 5.10. Overall architecture of Planaria 117

Figure 5.11. Overall workflow. 120

Figure 5.12. Throughput improvement over PREMA. 126

Figure 5.13. SLA satisfaction rate comparison. 126

Figure 5.14. Fairness improvement over PREMA. 127

Figure 5.15. Planaria energy reduction compared to PREMA. 128

Figure 5.16. Required number of nodes to achieve 99% SLA satisfaction. PREMA is not designed for SLA. To avoid unfairness, the results for PREMA is omitted. 129

Figure 5.17. Planaria improvements for single DNN inference compared to a conventional systolic accelerator with the same on-chip memory and compute resources. 129

Figure 5.18. Design space exploration for fission granularity. 132

Figure 5.19. Planaria power/area breakdown and its overheads. 134

Figure 6.1. Pruning operation on attention *Score*: (a) ideal magnitude-based pruning, (b) proposed differentiable pruning operation with soft threshold. 143

Figure 6.2. An example (a) attention layer sparsity and its corresponding pruning threshold values and (b) normalized training loss as fine-tuning epochs progress for BERT-L model on QNLI task from GLUE benchmark. 146

Figure 6.3. High-level overview of early-compute termination for dot-product operation $\mathcal{Q} \times \mathcal{K}^T$. In this example, $\mathcal{K}$ is represented in bit-serial format, whereas $\mathcal{Q}$ is in full-precision fixed-point format. 148

Figure 6.4. Overall microarchitecture of a LEOPARD tile. 150

Figure 6.5. A QK-DPU comprising (a) bit-serial dot-product engine, (b) margin calculation logic, (c) thresholding module, and (d) score index counter. 153

Figure 6.6. Accuracy before and after pruning-aware fine-tuning (prefix "G-": GLUE). We evaluate GPT-2 using perplexity, which favors a lower value. 158

Figure 6.7. Runtime pruning rate with LEOPARD. (prefix "G-": GLUE) 160

Figure 6.8. Cumulative pruning rate with respect to the number of bits processed during bit-serial early termination. Each line obtained by averaging across all the pruning rates per task. 162

Figure 6.9. Speedup comparison to baseline design for AE-LEOPARD and HP-LEOPARD (prefix "G-": GLUE dataset). 163

Figure 6.10. Total energy reduction for AE-LEOPARD and HP-LEOPARD compared to baseline (prefix "G-": GLUE dataset). 163

Figure 6.11. Normalized LEOPARD's average energy breakdown and the contribution of runtime pruning and bit-level early termination in energy saving (LEOPARD-P: with only pruning, and LEOPARD: pruning + bit-serial early termination) across one transformer head. 165

Figure 6.12. AE-LEOPARD: (a) layout (2.3×2.8 mm^2) and (b) area breakdown. 167

Figure 6.13. Back-end V-PU utilization over the QK-PU parallelism (N_{QK}). $N_{QK} = 6$ and $N_{QK} = 8$ form the favorable configurations in terms of back-end utilization in AE-LEOPARD and HP-LEOPARD, respectively. 168

Figure 6.14. Design space exploration for the resolution of bit-serial execution with respect to normalized average QK-DPU energy per *Score*. 168

LIST OF TABLES

Table 2.1. Evaluated benchmark DNNs 32

Table 2.2. BiHiwe and baselines platforms 32

Table 2.3. Accuracy before and after fine-tuning. 40

Table 3.1. Evaluated DNN models. 55

Table 3.2. Evaluated hardware platforms. 56

Table 4.1. Non-GEMM operators and their representative DNNs. 66

Table 4.2. Comparison of prior approaches for supporting non-GEMM operators with this work. † indicates that these aspects are supported partially. 71

Table 4.3. Non-GEMM examples and their implementations using primitives. 76

Table 4.4. Microarchitectural configurations of NPU-Tandem. 90

Table 5.1. Workload scenarios and benchmark DNNs from three domains: image classification [110, 241, 244, 116], object detection [162, 198, 199], and machine translation [274]. 123

Table 5.2. Layer sensitivity to various fission configurations. Each cell shows a configuration with its architectural attributes (parallelism, input activation reuse, partial sum reuse, and usage of omni-directional data movement) and the percentage of the layers that uses the configuration. 132

Table 6.1. Microarchitectural configurations of a LeOPArd tile. 155

Table 6.2. LeOPArd performance comparison under different scenarios with prior work [103, 261]. 165

ACKNOWLEDGEMENTS

To begin with, I express my deepest appreciation to my PhD advisor, Prof. Hadi Esmaeilzadeh. His unwavering support throughout the past five years helped me navigate through the highs and lows of my PhD journey. His constant belief in me, even during moments of self-doubt, has been invaluable. Beyond teaching me how to conduct top-notch research and shaping my scientific mindset, he has also influenced me to be a better individual. Without his guidance, I would not have accomplished all that I did during my PhD, nor would I have the potential for success in my future endeavors.

I would also like to extend my gratitude towards my thesis committee members Prof. Farinaz Koushanfar, Prof. Jose Martinez, Prof. Steven Swanson, and Prof. Dean Tullsen for their valuable feedback and insightful comments on my dissertation.

It was a great privilege for me to work under the mentorship of Prof. Nam Sung Kim at UIUC and Prof. Manya Ghobadi at MIT. They generously offered me invaluable guidance and advice for both my research and career path. I am truly grateful to Dr. Amir Yazdanbakhsh from Google Research, who not only acted as an exceptional mentor but also became a close friend, always offering his unwavering support for my research and career. My sincere thanks also go to Dr. Cliff Young, Dr. Mangpo Phothilimthana, Dr. Saurabh Kadekodi, Dr. Martin Maas, and Dr. Masoud Moshref, my mentors at Google Research, and to Colin Verrilli and Dr. Natarajan Vaidhyanathan, my mentors at Qualcomm. These outstanding researchers have played an essential role in my growth as a researcher and have instilled in me a mindset that enables me to apply my research effectively to real-world problems in industry settings.

My colleagues at Alternative Computing Technologies (ACT) lab, Dr. Byung Hoon Ahn, Sean Kinzer, Rohan Mahapatra, Hanyang Xu, Shu-Ting Wang, Parsa Asadi, Chris Priebe, Joon Kyung Kim, Lavanya Karthikeyan, Brahmendra Reddy Yatham, Fatemehsadat Mireshghallah, and preevious graduates Dr. Ahmed Taha Elthakeb, Prannoy Pilligundla, Dr. Hardik Sharma, Dr. Divya Mahajan, and Prof. Jongse Park have played an instrumental role during the course of my PhD. Their invaluable assistance and stimulating discussions not only facilitated my research

projects but also helped me overcome various obstacles along the way.

I feel incredibly grateful to have shared the experience of pursuing a PhD journey with my beloved wife, Mojan. Her unwavering dedication, support, and patience were integral to my success in the PhD program. I will never forget how she stood by me through the difficult times during the past five years, as well as provided me with invaluable insights, brainstorming, and constructive criticism regarding my research. The thought of starting and finishing this challenging path together fills me with immense happiness and pride, and I can't imagine having had a better source of encouragement and support than her.

Last but not least, I would like to express my deepest appreciation and gratitude to my family in Iran: my father Mahmoud, my mother Mitra, and my sisters Sara and Sorour. I couldn't have reached this point without their continuous sacrifices, love, and support. I am blessed to have them as my family and will forever be grateful to them for everything they have done for me.

The material in this dissertation is based on following listed papers.

Chapter 2 is a partial reprint of the material as it appears in: S. Ghodrati, H. Sharma, S. Kinzer, A. Yazdanbakhsh, J. Park, N. Kim, D. Burger, and H. Esmaeilzadeh, "Mixed-Signal Charge-Domain Acceleration of Deep Neural Networks through Interleaved Bit-Partitioned Arithmetic." in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2020. The dissertation author was the primary investigator and author of this paper.

Chapter 3 is a partial reprint of the material as it appears in: S. Ghodrati, H. Sharma, C. Young, N. Kim, and H. Esmaeilzadeh, "Bit-Parallel Vector Composability for Neural Acceleration." in *Design Automation Conference (DAC)*, 2020. The dissertation author was the primary investigator and author of this paper.

Chapter 4 is a partial reprint of the material as it appears in: S. Ghodrati, S. Kinzer, H. Xu, R. Mahapatra, Y. Kim, B. Ahn, D. Wang, L. Karthikeyan, A. Yazdanbakhsh, J. Park, N. Kim, and H. Esmaeilzadeh, "Tandem Processor: Grappling with Emerging Operators in Neural Networks." The dissertation author was the primary investigator and author of this paper.

Chapter 5 is a partial reprint of the material as it appears in: S. Ghodrati, B. Ahn, J. Kim, S. Kinzer, B. Yatham, N. Alla, H. Sharma, M. Alian, E. Ebrahimi, N. Kim, C. Young, and H. Esmaeilzadeh, “Planaria: Dynamic Architecture Fission for Spatial Multi-Tenant Acceleration of Deep Neural Networks.” in *International Symposium on Microarchitecture (MICRO)*, 2020. The dissertation author was the primary investigator and author of this paper.

Chapter 6 is a partial reprint of the material as it appears in: Z. Li, S. Ghodrati, A. Yazdanbakhsh, H. Esmaeilzadeh, M. Kang, “Accelerating Attention through Gradient-Based Learned Runtime Pruning.” in *International Symposium on Computer Architecture (ISCA)*, 2022. The dissertation author, Zheng Li, and Amir Yazdanbakhsh were the primary investigators and contributed equally to this paper.

This dissertation was in part supported by a Google PhD Fellowship, generous gifts from Samsung, Qualcomm, Microsoft, Xilinx as well as the National Science Foundation (NSF) awards CCF#2107598, CNS#1822273, National Institute of Health (NIH) award #R01EB028350, Defense Advanced Research Project Agency (DARPA) under agreement number #HR0011-18-C-0020, and Semiconductor Research Corporation (SRC) award #2021-AH-3039. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes not withstanding any copyright notation thereon. The views and conclusions contained herein are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied of Google, Qualcomm, Microsoft, Xilinx, Samsung, NSF, SRC, NIH, DARPA or the U.S. Government.

VITA

2017	Bachelor of Science in Electrical Engineering, Sharif University of Technology
2020	Master of Science in Computer Science, University of California San Diego
2023	Doctor of Philosophy in Computer Science, University of California San Diego

PUBLICATIONS

S. Ghodrati, S. Kinzer, H. Xu, R. Mahapatra, Y. Kim, B. Ahn, D. Wang, L. Karthikeyan, A. Yazdanbakhsh, J. Park, N. Kim, and H. Esmaeilzadeh. "Tandem Processor: Grappling with Emerging Operators in Neural Networks."

D. Wang, J. Lou, N. Jin, E. Mascarenhas, R. Mahapatra, S. Kinzer, **S. Ghodrati**, A. Yazdanbakhsh, H. Esmaeilzadeh, N. Kim. "MESA: Microarchitecture Extensions for Spatial Architecture Generation." In International Symposium on Computer Architecture (ISCA), 2023.

H. Esmaeilzadeh, **S. Ghodrati**, A. Kahng, J. Kim, S. Kinzer, S. Kundu, R. Mahapatra, S. Manasi, S. Sapatnekar, Z. Wang, and Z. Zeng. "Physically Accurate Learning-based Performance Prediction of Hardware-accelerated ML Algorithms." In Workshop on Machine Learning for CAD (MLCAD), 2022.

J. Kim, B. Ahn, S. Kinzer, **S. Ghodrati**, R. Mahapatra, B. Yatham, S. Wang, D. Kim, P. Sarikhani, B. Mahmoudi, D. Mahajan, J. Park, H. Esmaeilzadeh. "Yin-Yang: Programming Abstractions for Cross-Domain Multi-Acceleration." In IEEE Micro, 2022.

Z. Li*, **S. Ghodrati***, A. Yazdanbakhsh*, H. Esmaeilzadeh, M. Kang. "Accelerating Attention through Gradient-Based Learned Runtime Pruning." In International Symposium on Computer Architecture (ISCA), 2022. (* stands for equal contribution.)

H. Esmaeilzadeh, **S. Ghodrati**, J. Gu, S. Guo, A. Kahng, J. Kim, S. Kinzer, R. Mahapatra, S. Manasi, E. Mascarenhas, S. Sapatnekar, R. Varadarajan, Z. Wang, H. Xu, B. Yatham and Z. Zeng. "VeriGOOD-ML: An Open-Source Flow for Automated ML Hardware Synthesis." In International Conference on Computer Aided Design (ICCAD), 2021.

S. Kinzer, J. Kim, **S. Ghodrati**, B. Yatham, A. Althoff, D. Mahajan, S. Lerner, and H. Esmaeilzadeh. "A Computational Stack for Cross-Domain Acceleration." In International Symposium on High-Performance Computer Architecture (HPCA), 2021.

S. Ghodrati, B. Ahn, J. Kim, S. Kinzer, B. Yatham, N. Alla, H. Sharma, M. Alian, E. Ebrahimi, N. Kim, C. Young, and H. Esmaeilzadeh. "Planaria: Dynamic Architecture Fission for Spatial

Multi-Tenant Acceleration of Deep Neural Networks." In International Symposium on Microarchitecture (MICRO), 2020.

S. Ghodrati, H. Sharma, S. Kinzer, A. Yazdanbakhsh, J. Park, N. Kim, D. Burger, and H. Esmaeilzadeh. "Mixed-Signal Charge-Domain Acceleration of Deep Neural Networks through Interleaved Bit-Partitioned Arithmetic." In International Conference on Parallel Architectures and Compilation Techniques (PACT), 2020.

S. Ghodrati, H. Sharma, C. Young, N. Kim, and H. Esmaeilzadeh. "Bit-Parallel Vector Composability for Neural Acceleration." In Design Automation Conference (DAC), 2020.

A. Yazdanbakhsh, M. Brzozowski, B. Khaleghi **S. Ghodrati**, K. Samadi, N. Kim, and H. Esmaeilzadeh. "FlexiGAN: An End-to-End Solution for FPGA Acceleration of Generative Adversarial Networks." In International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2018.

ABSTRACT OF THE DISSERTATION

Architecting Machines for Green Intelligence

by

Soroush Ghodrati

Doctor of Philosophy in Computer Science

University of California San Diego, 2023

Professor Hadi Esmaeilzadeh, Chair

Deep learning has revolutionized the way humans interact with technology, enabling complex tasks that were once thought to be impossible. Its ability to process vast amounts of data and learn from patterns has led to significant advancements in various areas, such as image and speech recognition, natural language processing, and robotics. These advancements have had a significant impact on humans daily lives, from virtual assistants to self-driving cars, personalized recommendations on social media platforms, and fraud detection in banking and finance.

While deep learning has enabled remarkable advancements across various industries, its wide adoption has led to alarming repercussions in terms of carbon emissions and energy consumption. This is due to the high computational and storage requirements of deep learning

models, which result in the use of large data centers and computing infrastructures that consume a vast amount of energy and have become a significant contributor to carbon emissions. As deep learning models become increasingly complex, the energy consumption required for training and inference is expected to rise, exacerbating the problem. This becomes more crucial because the current computing infrastructures used for training and serving inference for these models are significantly underutilized.

This PhD dissertation sets out to take on this imperative challenge and rethink the design of custom neural accelerators and their adoption in both cloud and edge infrastructures by devising solutions across the whole compute stack ranging from circuits to systems and algorithms. To that end, the contributions of this dissertation are as follows:

- Devising BiHiwe, a programmable mixed-signal DNN accelerator that leverages the innate energy-efficiency of analog computing. To address the challenges associated with analog computing, I leverage the mathematical properties of deep learning operations and define a new computing model for dot-product operations along with its mixed-signal computing circuitry. I further design a programmable hierarchical clustered architecture to integrate the mixed-signal compute units and propose solutions to further mitigate the non-ideality in analog computing.

- Designing ultra-energy efficient acceleration solution for deeply quantized neural networks. The proposed design intersperses bit parallelism within data-level parallelism and dynamically interweaves the two together. This design paradigm enables dynamic composition of narrow-bitwidth vector engines at the bit granularity based on the required bitwidth of the DNN layers. This new composition mode amortizes the cost of aggregation and operand-delivery across a vector of elements and brings forth significant energy savings and performance improvements.

- Proposing a novel microarchitecture and Instruction Set Architecture for a companion processor in neural accelerators that tackles the challenges associated with executing emerging

and novel operations in DNNs. The design strikes a balance between customization and programmability to keep up with the volatility of deep learning research while offering significant performance and energy gains compared to prior work.

- Proposing Planaria, the very first neural accelerator design that offers simultaneous multi-tenant acceleration of DNNs. The design introduces and leverages the novel concept of runtime architecture fission, which breaks a monolithic accelerator into smaller yet full-fledged accelerators to enable spatial co-location of multiple DNN inference requests. To best utilize this microarchitecture capability, I also propose a task scheduling algorithm that breaks up the accelerator with respect to the current server load, DNN topology, and task priorities, all while considering the latency bounds of the tasks. This work opens a new dimension in the design of neural accelerators that considers utilization, cost-effectiveness, and responsiveness in datacenters.

- Devising a mathematical formulation for pruning the inconsequential operations in self-attention layers of transformer models. This formulation piggy backs on the back-propagation training to analytically co-optimize the threshold and the weights simultaneously, striking a formally optimal balance between accuracy and computation pruning. Additionally, I propose a bit-serial architecture, dubbed LEOPARD, to maximize the benefits by terminating computation even before pruning the following calculation without any approximation.

Chapter 1

Introduction

1.1 Motivation

The triumph of deep learning has led to remarkable achievements in the development of intelligent systems such as autonomous driving, robotics, and augmented/virtual reality. As a result, the demand for real-time AI services has been growing incessantly. On the other hand, the end of Dennard scaling [76] and the diminishing benefits from transistor scaling [83, 107] have instigated a new era of Domain-Specific Architectures or Accelerators. As such, accelerators have taken center stage in providing the high-performance computation required for these innovative applications in both cloud and edge systems.

Despite the rapid advances and widespread adoption of deep learning models and accelerators, there are some alarming consequences that have emerged, chief among them being the emission of carbon. The energy consumed in training and model selection for even a moderately-sized language model is comparable to the lifetime emissions of five US cars [237]. Furthermore, the energy consumption and cloud compute costs for these tasks have increased exponentially with the increasing complexity of the neural models. As shown in Figure 1.1, transformer-based NLP models have witnessed a 24,000-fold increase in these costs in less than two years [237]. Training a transformer model that is considered modest by today's standards requires a staggering 656 MWh and costs more than one million dollars. This becomes more worrisome when noting that the deep learning accelerators deployed in large-scale datacenters suffer from relatively

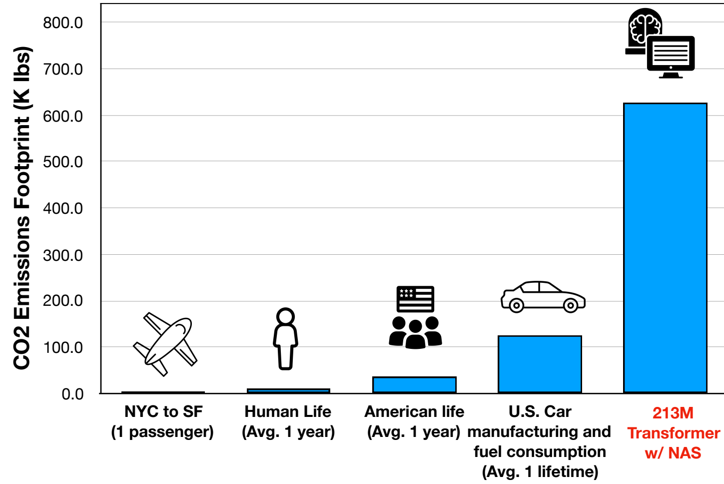


Figure 1.1. The trends in increasing the energy consumption and cloud compute cost of training emerging neural models.

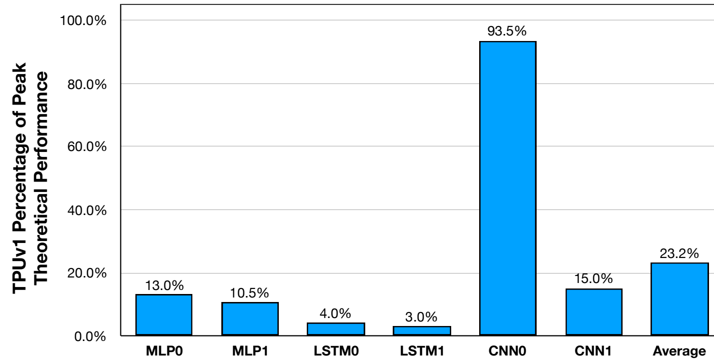


Figure 1.2. Theoretical peak performance utilization for Google TPUv1 [128].

low resource utilization. Figure 1.2 shows the percentage of peak performance achieved by the Google TPUv1 across various workloads. As shown, this datacneter accelerator can only achieve 23% of its peak performance, on average [128].

1.2 Contributions

These alarming repercussions call for rethinking the design of custom accelerators and their adoption in both cloud and edge solutions. *To that end, this PhD thesis focuses on two main objectives, i.e., 1) ultra-energy efficient computation and 2) cost-effective and green hardware utilization and devises solutions across the whole compute stack ranging from analog circuits*

to architectures, compilers, and datacenter systems. This dissertation not only resulted in intellectual insights for designing next-generation acceleration systems but has also yielded open-source hardware and software artifacts. Following discusses the contributions of this thesis:

Ultra energy-efficient neural acceleration via mixed-signal computing. Albeit low-power, mixed-signal circuitry suffers from significant overhead of Analog to Digital (A/D) conversion, limited range for information encoding, and susceptibility to noise. This thesis aims to address these challenges by offering and leveraging the following mathematical insight regarding vector dot-product—the basic operator in Deep Neural Networks (DNNs). *This operator can be reformulated as a wide regrouping of spatially parallel low-bitwidth calculations that are interleaved across the bit partitions of multiple elements of the vectors.* As such, the computational building block of the proposed accelerator becomes a wide bit-interleaved analog vector unit comprising a collection of low-bitwidth multiply-accumulate modules that operate in the analog domain and share a single A/D converter (ADC). This bit-partitioning results in a lower-resolution ADC while the wide regrouping alleviate the need for A/D conversion per operation, amortizing its cost across multiple bit-partitions of the vector elements. Moreover, the low-bitwidth modules require smaller encoding range and also provide larger margins for noise mitigation. I also utilize the switched-capacitor design for the bit-level reformulation of DNN operations. The proposed switched-capacitor circuitry performs the regrouped multiplications in the charge domain and accumulates the results of the group in its capacitors over multiple cycles. The capacitive accumulation combined with wide bit-partitioned regrouping reduces the rate of A/D conversions, further improving the overall efficiency of the design. With such mathematical reformulation and its switched-capacitor implementation, this dissertation defines one possible 3D-stacked microarchitecture, dubbed BiHiwe, that leverages clustering and hierarchical design to best utilize power-efficiency of the mixed-signal domain and 3D stacking. This thesis also builds models for noise, computational non-idealities, and variations due to usage of analog circuitry.

Interleaving data-level and bit-level parallelism for energy-efficient digital acceleration. Conventional neural accelerators rely on isolated self-sufficient functional units that perform an atomic operation while communicating the results through an operand delivery-aggregation logic. Each single unit processes all the bits of their operands atomically and produce all the bits of the results in isolation. This part of the thesis explores a different design style, where each unit is only responsible for a slice of the bit-level operations to interleave and combine the benefits of bit-level parallelism with the abundant data-level parallelism in deep neural networks. A dynamic collection of these units cooperate at runtime to generate bits of the results, collectively. Such cooperation requires extracting new grouping between the bits, which is only possible if the operands and operations are vectorizable. The abundance of Data-Level Parallelism and mostly repeated execution patterns, provides a unique opportunity to define and leverage this new dimension of Bit-Parallel Vector Composability. This design intersperses bit parallelism within data-level parallelism and dynamically interweaves the two together. As such, the building block of the proposed neural accelerator is a Composable Vector Unit that is a collection of Narrower-Bitwidth Vector Engines, which are dynamically composed or decomposed at the bit granularity. The comprehensive evaluation performed throughout this thesis shows significant performance and energy reduction benefits offered by this design style.

Balancing specialization and programmability for efficient end-to-end neural acceleration. Neural accelerators started with mostly focusing on GEneral Matrix Multiplication (GEMM) operations as they dominated neural network structures. However, as DNNs evolve they include more non-GEMM operations that are not only growing in variety themselves, but also are interwoven in diverse structures with the GEMM operations. At the beginning, the non-GEMM operations were limited to a rather small set of activation and pooling functions. To handle this limited set, it was natural to include a number of dedicated blocks or fall back to a general-purpose processor. With the structural evolution of DNNs, it is timely to revisit these design choices to both accommodate (1) newer structural varieties as well as (2) the diversity of the

operations. As such, this part of the dissertation sets out to explore the design of an on-chip companion, dubbed Tandem Processor, that complements the rather optimized GEMM unit in neural accelerators. This processor needs to be specialized to keep up with the GEMM unit; and yet needs to be programmable to address the (1) structural and (2) operational variations. To strike a balance between specialization and programmability, on the one hand, its memory access logic is specialized with a novel ISA/microarchitecture that alleviates the register file and its associated load/store operations. On the other hand, the calculations of the non-GEMM layers are only supported through primitive arithmetic/logic vector operations. Therefore, programmability is offered at the mathematical level. The enhancements due to the specialization of the memory access logic in the Tandem Processor and its tight integration with the GEMM unit sustain the throughput and the utilization of the neural accelerator. These design decisions are in contrast with the prior conventional approaches of using dedicated blocks and/or general-purpose processors. This thesis thoroughly evaluates the benefits of such processor design across various DNN benchmarks and design options.

Cost-effective accelerator utilization via spatial multi-tenancy. Cloud infrastructure and accelerators that offer INFerence-as-a-Service (INFaaS) have become the enabler of the rapid and diverse deployment of DNNs for various industries and markets. To that end, mostly accelerator-based INFaaS (Google's TPU [128], NVIDIA T4 [17], Microsoft Brainwave [90], etc.) has become the backbone of many real-life applications. However, as the demand for such services grows, merely scaling-out the number of accelerators is not economically cost-effective. Although multi-tenancy has propelled datacenter scalability, it has not been a primary factor in designing DNN accelerators due to the arms race for higher speed and efficiency. This part of the dissertation sets out to explore this timely requirement of multi-tenancy through a new dimension: dynamic architecture fission. To that end, I define Planaria[1] that can *dynamically fission (break)* into multiple smaller yet full-fledged DNN engines at runtime. This microarchitectural capability

[1]Planaria is a species which, when an individual is cut (*fissioned*) into pieces, all pieces can regenerate to fully formed individuals.

enables *spatially co-locating* multiple DNN inference services on the same hardware, offering simultaneous multi-tenant DNN acceleration. To realize this dynamic reconfigurability, this thesis first devises breakable omni-directional systolic arrays for DNN acceleration that allows omni-directional flow of data. Second, it uses this capability and a unique organization of on-chip memory, interconnection, and compute resources to enable fission in systolic array based DNN accelerators. Architecture fission and its associated flexibility enables an extra degree of freedom for task scheduling, that even allows breaking the accelerator with regard to the server load, DNN topology, and task priority. As such, it can simultaneously co-locate DNNs to enhance utilization, throughput, QoS, and fairness. Comprehensive comparisons against a design that offers multi-tenancy through time-multiplexing the accelerator across multiple tasks, show significant benefits in terms of throughput, Service Level Agreement (SLA) satisfaction rate, fairness, and energy reduction.

Energy-efficient inference acceleration of large-scale transformer models. Self-attention is a key enabler of state-of-art accuracy for various transformer-based Natural Language Processing models. This attention mechanism calculates a correlation score for each word with respect to the other words in a sentence. Commonly, only a small subset of words highly correlates with the word under attention, which is only determined at runtime. As such, a significant amount of computation is inconsequential due to low attention scores and can potentially be pruned. The main challenge is finding the threshold for the scores below which subsequent computation will be inconsequential. Although such a threshold is discrete, this thesis formulates its search through a soft differentiable regularizer integrated into the loss function of the training. This formulation piggy backs on the back-propagation training to analytically co-optimize the threshold and the weights simultaneously, striking a formally optimal balance between accuracy and computation pruning. To best utilize this mathematical innovation, my dissertation devises a bit-serial architecture, dubbed LEOPARD, for transformer models with bit-level early termination microarchitectural mechanism. Post-layout results show that LEOPARD yields significant gains

in speedup and energy reduction, while keeping the average accuracy virtually intact ($< 0.2\%$ degradation).

1.3 Acknowledgements

This chapter is, in part, a reprint of the following publications: (1) S. Ghodrati, H. Sharma, S. Kinzer, A. Yazdanbakhsh, J. Park, N. Kim, D. Burger, and H. Esmaeilzadeh, "Mixed-Signal Charge-Domain Acceleration of Deep Neural Networks through Interleaved Bit-Partitioned Arithmetic." in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2020. (2) S. Ghodrati, H. Sharma, C. Young, N. Kim, and H. Esmaeilzadeh, "Bit-Parallel Vector Composability for Neural Acceleration." in *Design Automation Conference (DAC)*, 2020. (3) S. Ghodrati, S. Kinzer, H. Xu, R. Mahapatra, Y. Kim, B. Ahn, D. Wang, L. Karthikeyan, A. Yazdanbakhsh, J. Park, N. Kim, and H. Esmaeilzadeh, "Tandem Processor: Grappling with Emerging Operators in Neural Networks." (4) S. Ghodrati, B. Ahn, J. Kim, S. Kinzer, B. Yatham, N. Alla, H. Sharma, M. Alian, E. Ebrahimi, N. Kim, C. Young, and H. Esmaeilzadeh, "Planaria: Dynamic Architecture Fission for Spatial Multi-Tenant Acceleration of Deep Neural Networks." in *International Symposium on Microarchitecture (MICRO)*, 2020. (5) Z. Li, S. Ghodrati, A. Yazdanbakhsh, H. Esmaeilzadeh, M. Kang, "Accelerating Attention through Gradient-Based Learned Runtime Pruning." in *International Symposium on Computer Architecture (ISCA)*, 2022. The dissertation author was the (co)primary investigator of these papers.

Chapter 2

Ultra-Energy Efficient DNN Acceleration via Mixed-Signal Computing

2.1 Introduction

With the diminishing benefits from general-purpose processors [108, 258, 84, 290], there is an explosion of digital accelerators for DNNs [86, 58, 92, 73, 168, 295, 29, 129, 221, 64, 184, 30, 106, 55, 56, 135, 128, 54, 222, 27, 112, 152]. Mixed-signal acceleration [215, 233, 253, 158, 37, 150, 39, 44, 234, 294, 151] is also gaining traction. Albeit low-power, mixed-signal circuitry suffers from limited range of information encoding, is susceptible to noise, lacks fine-grained control mechanism and imposes significant overheads for Analog to Digital (A/D) conversions. As a point of reference, for an 8-bit×8-bit MACC which produces a 16-bit output at 500 Mhz, A/D conversion costs about 1,000× higher energy than the MACC itself at 45 nm. In addition, encoding 256 levels for 8-bit inputs in less than 1 Volt allocates 3.9 mV for each level, significantly restricting both the representation capabilities as well as the noise margins. This work sets out to address these challenges by inspecting the mathematical foundation of deep neural networks and makes the following contributions.

(1) This research offers and leverages the insight that the set of MACC operations within one vector dot-product can be partitioned, interleaved, and regrouped at the *bit level* without affecting the mathematical integrity of dot-product. Unlike prior work [233, 151, 59], this work does not rely on changing the mathematics of the computation to enable mixed-

signal acceleration. Namely, PRIME [59] leverages memristive technology to enable analog computation, but relies on several truncations during computations of intermediate data to overcome the overheads of A/D conversions. In contrast, this work only rearranges the bit-wise arithmetic calculations across multiple elements of the vectors to utilize a group of lower bitwidth analog units for higher bitwidth operations. The key insight is that a binary value can be expressed as the sum of products similar to dot-product, which is also a sum of multiplications ($a = \vec{X} \bullet \vec{W} = \sum_i x_i \times w_i$). Each x_i or w_i can be expressed as $\sum_j (2^j \times b_j)$ where b_js are the individual bits or as $\sum_j (2^{4j} \times bp_j)$, where bp_js are 4-bit partitions for instance. Our interleaved arithmetic utilizes the *distributive and associative property* of multiplication and addition at the *bit granularity* for partitioning and regrouping.

The proposed model, first, bit-partitions all elements of the two vectors, and then *distributes* the MACC operations of the dot-product over these bit partitions. Then, our mathematical formulation exploits the *associative property* of the multiply and add to group and co-locate bit-partitions that are at the same significance position. This significance-based rearrangement enables factoring out the power-of-two multiplicand that signifies the position of the bit-partitions. The factoring enables regrouping the partial results from a set of lower-bitwidth MACCs as one spatially parallel operation in the analog domain, while the group shares a *single* A/D converter (ADC). The power-of-two multiplicand will be applied later digitally to the accumulated result of the group operation. To this end, we reformulate vector dot-product as a wide regrouping of interleaved and bit-partitioned operations across multiple elements of the two vectors (see section 2.2). This spatial regrouping of operations and parallel execution is in contrast with prior analog-based accelerators such as PRIME [59] and ISAAC [215], which although use bit-partitioning but perform MACC operations serially over multiple cycles on bit (partitions) of the operands or RedEye [158] that does not exploit any sort of bit-partitioning.

The bit-partitioning lowers the resolution of ADCs while the wide regrouping amortizes the cost of each A/D conversion across multiple bit-partitions of the vector elements. Using low-bitwidth operands for analog MACCs also provides a larger headroom between the value

encoding levels in the analog domain. The headroom tackles the limited range of encoding and offers more robustness to noise, an inherent non-ideality in the analog mode.

(2) At the circuit level, the accelerator is designed using switched-capacitor circuitry that stores the partial results as electric charge over time without conversion to the digital domain at each cycle. The low-bitwidth MACCs are performed in charge domain with a set of charge-sharing capacitors. This design choice lowers the rate of A/D conversion as it implements accumulation as a gradual storage of charge in a set of parallel capacitors. These capacitors not only aggregate the result of a group of low-bitwidth MACCs, but also enable accumulating results over time. As such, the architecture enables dividing the longer vectors into shorter sub-vectors that are multiply-accumulated over time with a single group of spatially parallel low-bitwidth MACCs. The results are accumulated over multiple cycles in the group's capacitors. Because the capacitors can hold the charge from cycle to cycle, the A/D conversion is not necessary in each cycle. This reduction in rate of A/D conversion is in addition to the amortized cost of ADCs across the analog low-bitwidth MACCs of the group (see section 2.3).

(3) We take a systematic approach and perform a step-by-step analysis to evaluate the contribution of each technique in tackling the challenges of mixed-signal design and maximizing its benefits. This analysis shows that Interleaved Bit-Partitioning is the most effective technique. On one hand, Spatially Wide Regrouping is the second most effective technique in improving area efficiency of the arithmetic operations that enables integrating more in a given area, improving design parallelism. The benefit stems from sharing a single ADC across the groupings of the low-bitwidth analog MACC units, amortizing its area. On the other hand, Charge-Domain Computation ranks second in improving the power efficiency that is the fruit of reducing the rate of the A/D conversions, through accumulation and storage of the intermediate results in capacitors.

With these insights, we devise a hierarchical 3D-stacked instance of the microarchitecture, named BiHiwe, that leverages the proposed arithmetic and building blocks, yet offers

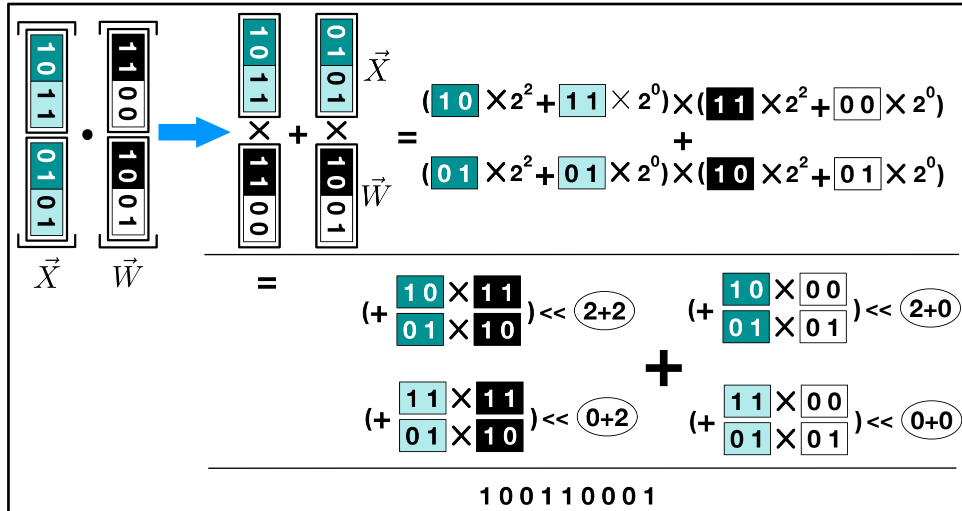
programmability and domain generality. Evaluating this carefully balanced design of BiHiWe with a diverse set of ten DNN benchmarks shows that BiHiWe delivers 5.5×speedup over the purely digital 3D-stacked DNN accelerator, Tetris [92], with only 0.5% loss in accuracy achieved after mitigating noise, computation error, and Process-Voltage-Temperature (PVT) variations. With 8-bit execution, BiHiWe offers 35.4×and 70.1×higher Performance-per-Watt compared to RTX 2080 TI and Titan Xp, respectively. Compared to the mixed-signal CMOS RedEye [158], memristive ISAAC [215] and PipeLayer [227], BiHiWe delivers 5.5×, 3.6×, and 9.6× higher Performance-per-Watt, respectively. With these benefits, this work marks an initial effort to use mathematical insights for devising mixed-signal DNN accelerators.

2.2 Wide, Interleaved, and Bit-Partitioned Arithmetic

A key idea of this work is the mathematical insight that enables utilizing low bitwidth mixed-signal units in spatially parallel groups.

Bit-Level partitioning and interleaving of MACCs. To further detail the proposed mathematical reformulation, Figure 3.2(a) delves into the bit-level operations of dot-product on vectors with 2-elements containing 4-bit values. As illustrated with different colors, each 4-bit element can be written in the form of sum of 2-bit partitions multiplied by powers of 2 (shift). As discussed, vector dot-product is also a sum of multiplications. Therefore, by utilizing the distributive property of addition and multiplication, we can rewrite the vector dot-product in terms of the bit partitions. However, we also leverage the associativity of the addition and multiplication to regroup the bit-partitions that are in the same positions, together. For instance, in Figure 3.2, the black partitions that represent the Most Significant Bits (MSBs) of the $\vec{W}$ vector are multiplied in parallel to the teal[1] partitions, representing the MSBs of the $\vec{X}$. Because of the distributivity of multiplication, the shift amount of (2+2) can be postponed after the bit-partitions are multiply-accumulated. The different colors of the boxes in Figure 3.2 illustrates the interleaved regrouping of the bit-partitions. Each group is a set of spatially parallel bit-partitioned MACC operations

[1]Color teal in Figure 3.2 is the darkest gray in black and white prints.



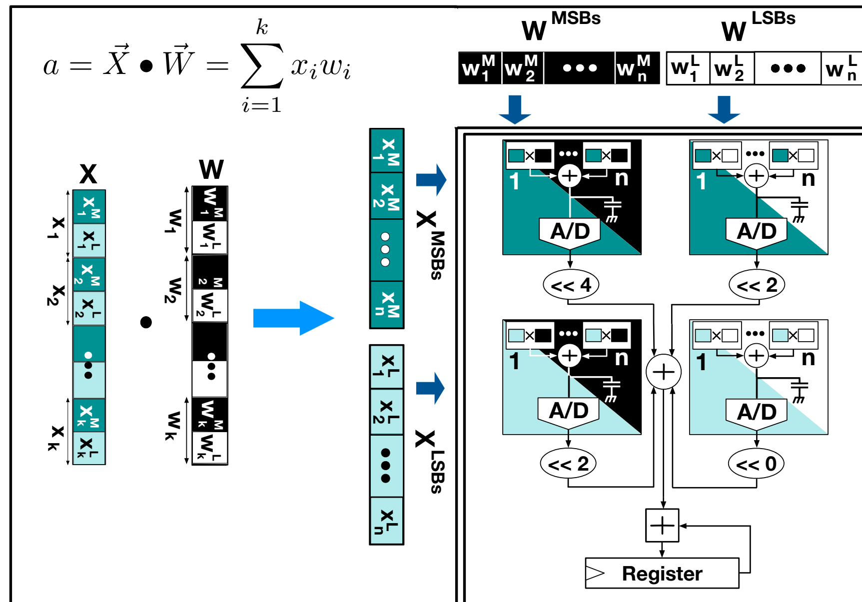
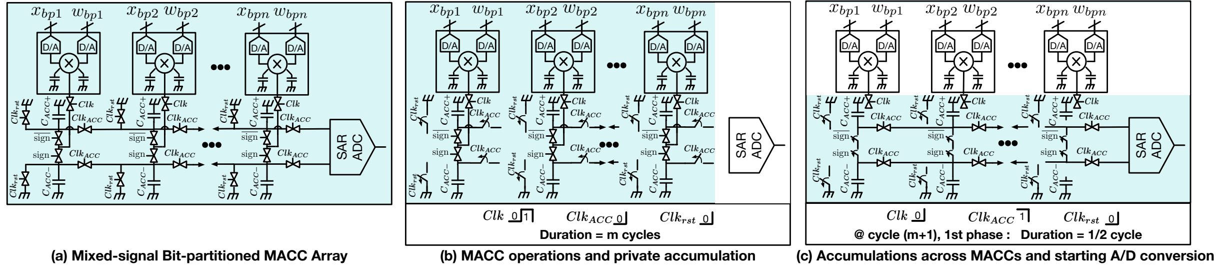


Figure 2.1. Wide, interleaved, and bit-partitioned mathematical formulation.

that are drawn from different elements of the two vectors. The low-bitwidth nature of these operations enables execution in the analog domain without the need for A/D conversion for each individual bit-partitioned operation. As such, our proposed reformulation amortizes the cost of A/D conversion across the bit-partitions of different elements of the vectors as elaborated below.

Wide, interleaved, and bit-partitioned vector dot-product. Figure 3.2(b) illustrates the proposed vector dot-product operation with 4-bit elements that are bit partitioned to 2-bit sub-elements. For instance, as illustrated, the elements of vector X, denoted as x_i, are first bit partitioned to x_i^L and x_i^M. The former represents the two Least Significant Bits (LSBs) and the latter represents the Most Significant Bits (MSBs). Similarly, the elements of vector W are also bit partitioned to the w_i^L and w_i^M sub-elements. Then, each vector (e.g., W) is rearranged into two bit-partitioned sub-vectors, W^{LSBs} and W^{MSBs}. In the current implementations of BiHiwe architecture, the size of bit-partitioning is fixed across the entire architecture. Therefore, the rearrangement is just rewiring the bits to the compute units that imposes modestly minimal overhead (less than 1%). Figure 3.2 is merely an illustration and there is no need for extra storage or movement of elements. As depicted with color coding, after the rewiring, W^{LSBs} represents all the least significant bit-partitions from different elements of vector W, while the MSBs are rewired in W^{MSBs}. The same rewiring is repeated for the vector X. This rearrangement, puts all the bit-partitions from all the elements of the vectors with the same significance in one group, denoted as W^{LSBs}, W^{MSBs}, X^{LSBs}, X^{MSBs}. Therefore, when a pair of the groups (e.g., X^{MSBs} and W^{MSBs} in Figure 3.2(c)) are multiplied to generate the partial products, (1) the shift amount ("$\ll 4$" in this case) is the same for all the bit-partitions and (2) the shift can be done after partial products from different sub-elements are accumulated together.

As shown in Figure 3.2(c), the low-bitwidth elements are multiplied together and accumulated in the analog domain. Accumulation in the digital domain would require an adder tree which is costly compared to the analog accumulation that merely requires connectivity between the multiplier outputs. It is only after several analog multiply-accumulations that the results are



(a) Mixed-signal Bit-partitioned MACC Array (b) MACC operations and private accumulation (c) Accumulations across MACCs and starting A/D conversion

Figure 2.2. MS-BPMacc and its operational modes.

converted to digital for shift and aggregation with partial products from the other groups. This is not only because of spatially wide grouping of the low-bitwidth MACC operations, but also, as will be discussed in the next section, due to the accumulation of the partial results in the analog domain by storing electric charge in capacitors before ADCs (see Figure 3.2(c)). If the size of vectors exceeds the predefined value of (size of spatially low-bitwidth array) × (number of capacitive accumulation cycles), these converted partial results will be added up in the digital domain using a register. For this pattern of computation, we are effectively utilizing the *distributive and associative property* of multiplication and addition for dot-product but at the *bit granularity*. This rearrangement and spatially parallel (i.e., wide) bit-partitioned computation is in contrast with temporally bit-serial digital [73, 129, 152, 217] and analog [215] DNN accelerators.

2.3 Switched-Capacitor Circuit Design for Interleaved Bit-Partitioning

To exploit the aforementioned arithmetic, an analog vector unit needs to be designed. This building block is a collection of low-bitwidth analog MACCs that operate in parallel on sub-elements from the two vectors under dot-product. This wide structure is dubbed Mixed-Signal Bit-Partitioned MACC Array (MS-BPMACC). Within the MS-BPMACC, we design the low-bitwidth MACC units using switched-capacitor circuitry [253, 37, 294, 151, 101], implementing the MACC operations in the charge-domain rather than using resistive-ladders to compute in current domain [215, 234, 59]. Compared to the current-domain approach, switched-capacitors (1) enable result accumulation in the analog domain by storing them as electric

charge, eliminating the need for A/D conversion at every cycle, and (2) make the relative ratio of capacitors the determining factor in analog multiplication. Dependence to ratio and not the absolute sizes makes the design more resilient to process variation.

2.3.1 Mixed-Signal Bit-Partitioned MACC Array

Figure 2.2(a) depicts an array of n low-bitwidth MACCs, constituting the $\mathcal{MS}$-BPMACC unit, which perform operations for m cycles in the analog domain. Each low-bitwidth MACC unit receives a pair of bit-partitions (x_{bpi}, w_{bpi}) from the sub-vectors. These bit-partitions are fed to Digital to Analog (D/A) converters to enable charge-domain MACC operations. Low-bitwidth MACC units are equipped with their own pair of accumulating capacitors (C_{ACC+}, C_{ACC-}), which perform the accumulation over time across multiple sub-vectors. The pair is used to handle positive and negative values by accumulating them separately on one or the other capacitor. Figure 2.2(b) illustrates the MACC computation mode of the $\mathcal{MS}$-BPMACC unit. Over m cycles, each low-bitwidth MACC unit works separately and accumulates the partial results privately on its own pair of C_{ACC}s. To enable the private accumulation mode the transmission gates between different MACC units are all disconnected (shown with open switches) and only the capacitors' private transmission gates are connected. Aggregation across the multiple low-btiwidth MACCs happens in the first half of cycle $m+1$, shown in Figure 2.2(c) . In this half cycle, the private results get aggregated across all n MACC units within the $\mathcal{MS}$-BPMACC. The transmission gates between the capacitors connect them and a simple charge sharing between the capacitors yields the aggregated result of $m \times n$ number of multiply-adds. Clk_{ACC} is the control signal which connects the C_{ACC}s. This aggregation happens for both positive and negative values (across C_{ACC+} s and C_{ACC-} s respectively) at the same time. The *single* ADC in the $\mathcal{MS}$-BPMACC is responsible for converting the aggregated result, which also starts at the first stage of cycle $m+1$. The accumulating capacitors (C_{ACC}s), are connected to a Successive Approximation Register (SAR) ADC and share their stored charge with the Sample and Hold block (S&H) of the ADC. This (S&H) block has differential inputs which samples the positive

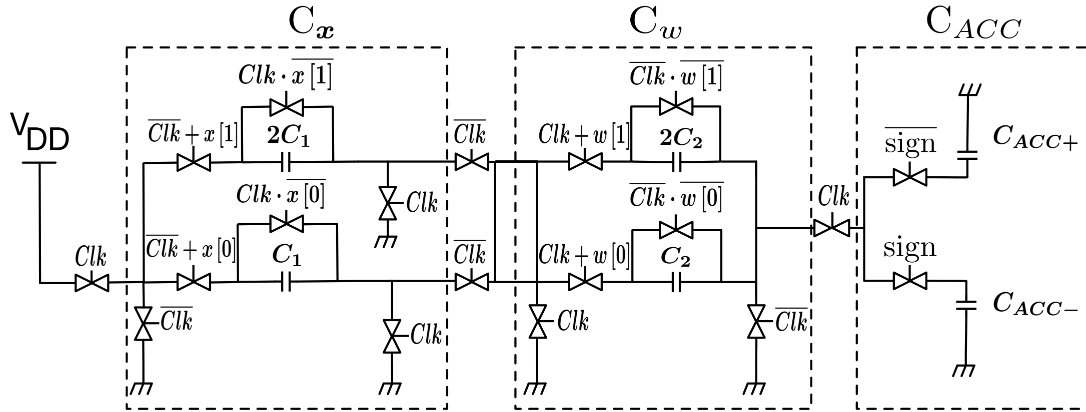


Figure 2.3. Low-bitwidth switched-capacitor MACC.

and negative results separately and holds them for the process of A/D conversion. In the second phase of cycle $m+1$ all the $\mathrm{C_{ACC}}$s get disconnected from the ADC and Clk_{rst} connects them to the ground to clear their charge for the next iteration of wide, bit-interleaved calculations. There is a trade-off between resolution and sample rate of ADC, which also defines its topology. For instance, Flash ADCs are suitable for high sample rate but low resolution designs. SAR ADC is a better choice when it comes to medium resolution (8-12 bits) and sample rate (1-500 Mega-Samples/sec). We choose a 10-bit, 15 Mega-Samples/sec SAR ADC [109] as it strikes the best balance between rate and resolution for $\mathcal{MS}$-BPMACCs based on design space exploration shown in Figure 2.17. The process of A/D conversion takes $m+1$ cycles, pipelined with vector dot-products.

The $\mathcal{MS}$-BPMACC computes the low-bitwidth MACC operations in charge-domain as the following discusses.

2.3.2 Low-Bitwidth Switched-Capacitor MACC

Figure 2.3 depicts the design of a single 3-bit sign-magnitude MACC. The $x_s x_1 x_0$ and $w_s w_1 w_0$ denote the bit-partitions operands. The result of each MACC operation is retained as electric charge in the accumulating capacitor ($\mathrm{C_{ACC}}$). In addition to $\mathrm{C_{ACC}}$, the MACC unit contains two capacitive Digital-to-Analog Converters (DACs), one for inputs ($\mathrm{C_X}$) and one for weights ($\mathrm{C_W}$). The $\mathrm{C_X}$ and $\mathrm{C_W}$ convert the 2-bit magnitude of the input and weight to the

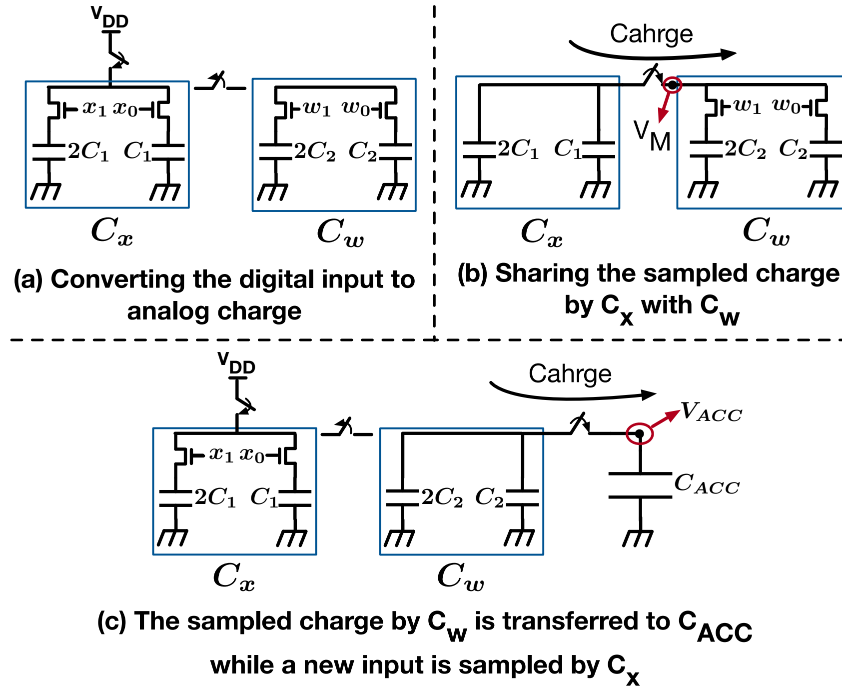


Figure 2.4. Charge-domain MACC; phase by phase.

analog domain as an electric charge proportional to $|x|$ and $|w|$ respectively. C_X and C_W are each composed of two capacitors ((C_1, $2C_1$) for C_X and (C_2, $2C_2$) for C_W) which operate in parallel and are combined to convert the operands to analog domain. Each of these capacitors are controlled by a pair of transmission gates which determine if a capacitor is active or inactive. Another set of transmission gates connect the two D/A converters and share charge when partitions of x and w are multiplied. The resulting shared charge is stored on either C_{ACC+} or C_{ACC-} depending on the sign control signal produced by $x_s \oplus w_s$. During multiplication, the transmission gates are coordinated by a pair of complimentary non-overlapping clock signals, Clk and $\overline{Clk}$.

Charge-domain MACC. Figure 2.4 shows the phase-by-phase process of a MACC, the phases of which are described below.

$Clk_{\phi(1)}$**:** The first phase (Figure 2.4(a)) consists C_X converting digital input (x) to a charge proportional to its magnitude. Since, the sampled charge (Q_{sx}) by C_X in the first phase is equal to:

$$Q_{sx} = v_{DD} \times (|X|C_1) \tag{2.1}$$

$\overline{Clk}_{\phi(2)}$**:** In the second phase (Figure 2.4(b)), the multiplication happens via a charge-sharing process between C_X and C_W. The 2-bit partition of the weight is applied to C_W and sets its equivalent capacitance to $|w|C_2$. At the same time, the C_X redistributes its sampled charge (Q_{sx}) over all of its capacitors ($3 \times C_1$) as well as the equivalent capacitor of C_W. The voltage (V_M) at the junction of C_X and C_W is as follows:

$$V_M = \frac{Q_{sx}}{C_{tot}} = \frac{v_{DD} \times (|X|C_1)}{3C_1 + |w|C_2} \tag{2.2}$$

Because the sampled charge is shared with the weight capacitors, the stored charge (Q_{sw}) on C_W is equal to:

$$Q_{sw} = V_M \times |w|C_2 = |x| \times |w| \left(\frac{C_2 C_1 v_{DD}}{3C_1 + |w|C_2} \right) \tag{2.3}$$

Equation 2.3 shows that Q_{sw} is proportional to $|x| \times |w|$, but includes a non-linearity term in the denominator ($|w|$). To mitigate that C_1 must be much larger than C_2. Further mitigation is considered as discussed in Section 2.6. With this choice, Q_{sw} becomes $|x| \times |w| \frac{C_2 v_{DD}}{3}$.

$Clk_{\phi(3)}$**:** In the last phase, (Figure 2.4(c)), the charge from multiplication is shared with C_{ACC} for accumulation. The sign bits (x_s and w_s) determine which of C_{ACC+} or C_{ACC-} is selected for accumulation. The sampled charge by $|w|C_2$ is then redistributed over the selected C_{ACC} as well as all the capacitors of C_W ($= 3C_2$). Theoretically, C_{ACC} must be infinitely larger than $3C_2$ to completely absorb the charge from multiplication. However, in reality, some charge remains unabsorbed, leading to a pattern of computational error, which is mitigated as discussed in Section 2.6. Ideally, the V_{ACC} voltage on C_{ACC} is:

$$V_{ACC} = |x||w| \left(\frac{C_2 v_{DD}}{3 \times C_{ACC}} \right) \tag{2.4}$$

While the charge sharing and accumulation happens on C_{ACC}, a new input is fed into C_X, starting a new MACC process in a pipelined fashion. This process repeats for all low-bitwidth MACC units over multiple cycles before one A/D conversion.

2.4 Mixed-Signal Architecture Design for Spatial Bit-Partitioning

Last section provided the detailed innards of low-bitwidth MACCs and how they can be used to construct a *low-bitwidth* spatially interleaved dot-product unit ($\mathcal{MS}$-BPMACC). This section, focuses on architecting a *higher bitwidth* dot-product engine, called $\mathcal{MS}$-WAGG, from a collection of $\mathcal{MS}$-BPMACCs. This engine is named $\mathcal{MS}$-WAGG as it is a Mixed- Signal Wide Aggregator that operates on bit-partitioned vectors in SIMD fashion. Instead of just describing the design, we take a quantitive journey that step-by-step highlights how much each design decision contributes to improving the power and area efficiency. Finally, we elaborate on how to utilize this engine to construct a full-fledged programmable mixed-signal DNN accelerator.

2.4.1 Mixed-Signal Wide Aggregator

To better understand the tradeoffs in designing $\mathcal{MS}$-WAGG, we contrast it with a basic mixed-signal dot-product engine, called $\mathcal{MS}$-BASIC, that does not utilize bit-partitioning (see Figure 2.5). Consequently, the D/A converters in Figure 2.5 are stained with two different shades of a color to highlight that all of the different bit-partitions of each operand are kept together. Each analog multiplier receives all operands'bits and converts the multiplication result to digital domain to go through an adder tree. Mixed-signal DNN accelerators are essentially an optimized transformation of this basic engine. Here, we discuss how much each of our innovations contributes to the design transformation that yields $\mathcal{MS}$-WAGG. Figure 2.6 illustrates a possible $\mathcal{MS}$-WAGG design, comprising 16 $\mathcal{MS}$-BPMACCs, necessary to perform 8-bit by 8-bit vector dot-product with 2-bit partitioning[2]. In contrast to the $\mathcal{MS}$-BASIC, each

[2]2-bit partitioning is the optimal choice (design space exploration in Figure 2.15).

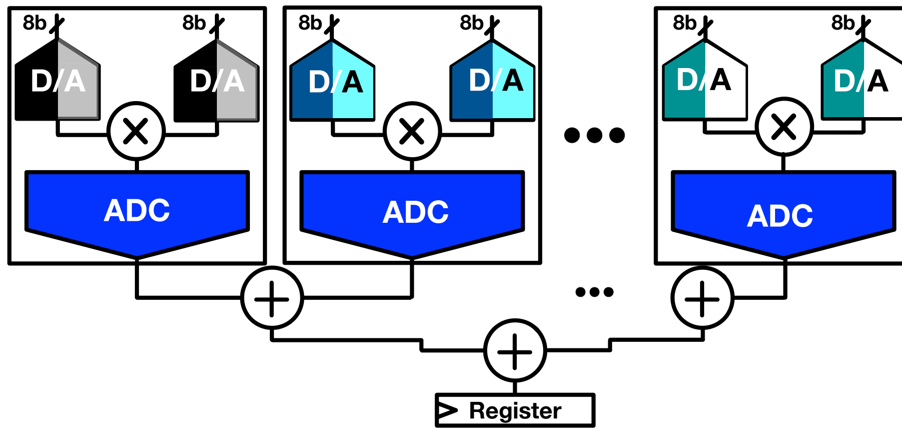


Figure 2.5. Basic mixed-signal dot-product engine.

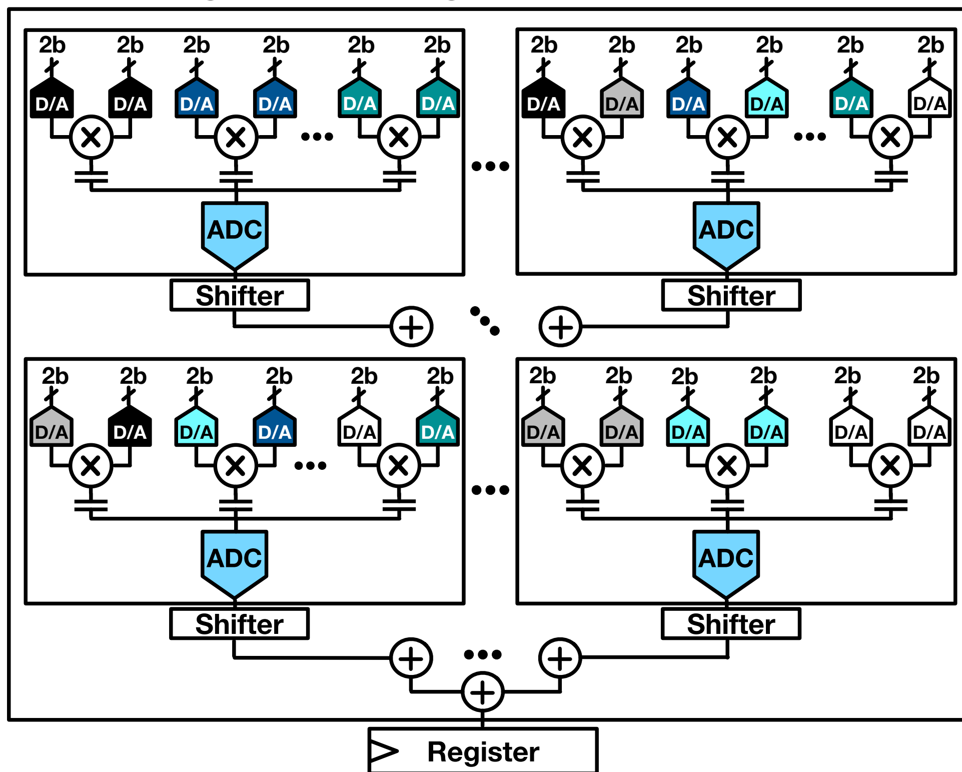


Figure 2.6. Our mixed-signal dot-product engine ($\mathcal{MS}$-WAgg).

D/A converter in Figure 2.6 is colored with one shade to show each input is just a bit-partition. In this case, the number of $\mathcal{MS}$-BPMACCs, 16 (=*four*×*four*), comes from the fact that each of the two 8-bit operands can be partitioned to *four* 2-bit values. Each of the *four* 2-bit partitions of the multiplicand need to be multiply-accumulated with all the multiplier's *four* 2-bit partitions. As discussed in Section 2.2, each $\mathcal{MS}$-WAGG also performs the necessary shift operations to combine the low-bitwidth results from its 16 $\mathcal{MS}$-BPMACCs. By aggregating the partial results of each $\mathcal{MS}$-BPMACC in the digital domain, the $\mathcal{MS}$-WAGG engine generates a scalar which is stored on its output register.

2.4.2 $\mathcal{MS}$-WAGG Design Decisions and Tradeoffs

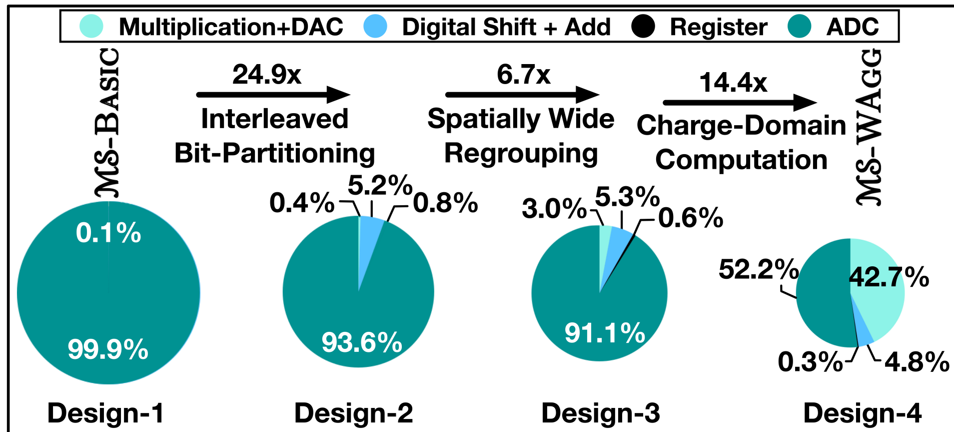
The design of $\mathcal{MS}$-WAGG stems from three main techniques: (1) Interleaved Bit-Partitioning, (2) Spatially Wide Regrouping, and (3) Charge-Domain Computation. For all the analyses in this section, 500 Mhz frequency at 45 nm is used to design an 8-bit vector dot-product engine. Figure 2.7(a) and (b) illustrates the contribution of each technique in power and area improvement, respectively. Improving area efficiency has a direct effect on performance as it enables integrating more compute units in a given area, improving design parallelism. The pie charts show how much of the total power/area is consumed by each of hardware components: analog multiplication, digital shift-and-add logic, register, ADC. D/A conversion is part of the analog multiplier as discussed in Section 2.3. The size of the pie is pictorially reduced to show that the total power/area is decreasing. The first pie chart belongs to $\mathcal{MS}$-BASIC–merely a point of reference–that performs an 8-bit×8-bit MACC in the analog domain and converts the 16-bit result to digital, while the last chart is of $\mathcal{MS}$-WAGG. The following discusses each technique and its effects on power/area efficiency.

(1) Interleaved Bit-Partitioning is the most effective technique in improving both power and area of the mixed-signal dot-product engines. This technique partitions each operand to lower bitwidth suboperands, and then interleaves the bit-partitions. Interleaved Bit-Partitioning enables replacing the 8-bit×8-bit MACC and its high resolution (16-bit) ADC in $\mathcal{MS}$-BASIC

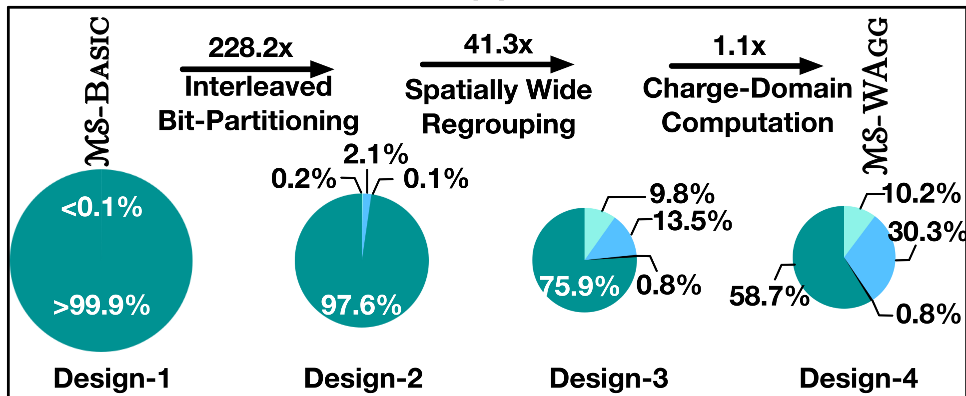
with 16 2-bit×2-bit MACCs and significantly lower resolution (4-bit) ADCs . By applying this technique, the power and area for an 8-bit MACC operation improves by 24.9× and 228.2×, respectively (Comparing Design-2 with Design-1 in Figure 2.7). This improvement stems from the fact that power and area of ADC increases dramatically with its resolution.

(2) Spatially Wide Regrouping is the second technique that regroups a wide array of interleaved lower bitwidth MACC units to share a single ADC. The outputs of lower-bitiwidth MACC units is aggregated in the analog domain, the result of which is fed to the ADC. This technique ranks second in improving area efficiency (41.3× when comparing Design-3 with Design-2 in Figure 2.7(b)). Sharing a single ADC across a wide group of lower-bitwidth MACC units, reduces the effective number of ADCs, leading to lower area. This sharing increases 4-bit resolution of the ADCs to 7-bits (sharing an ADC with 8 2-bit×2-bit MACCs) as more number of low-bitwidth MACC operations are aggregated in the analog domain before conversion; however, this increase in the ADC's power/area is sub-exponential. The benefit comes from the fact that Spatially Wide Regrouping enables shifting the ADC design style from Flash to Pipelined or SAR in the same frequency. Exploiting this technique also improves the power efficiency by 6.7×.

(3) Charge-Domain Computation is the second most effective technique in improving power-efficiency. Accumulating the partial results as electric charge in capacitors eliminates the necessity of A/D conversion at each cycle, leading to significant power reduction. This additional accumulation in the analog domain requires higher resolution ADCs (10-bits); however, the reduced rate of the A/D conversion trumps the resolution increase. This technique yields an additional 14.4× improvement in power efficiency (Design-4 vs Design-3 in Figure 2.7(a)). The number of the ADC remains the same but the reduced rate enables choosing an ADC with lower sample rate. Lower sample rate ADCs require lower-area subcomponents that can reduce its overall area. However, the increase in resolution counteracts this benefit to a large degree. As such, this technique only reduces the area by 1.1× (Design-4 compared to Design-3 in Figure 2.7(b)).



(a)



(b)

Figure 2.7. Step-by-step analysis of improvement in (a) power and (b) area.

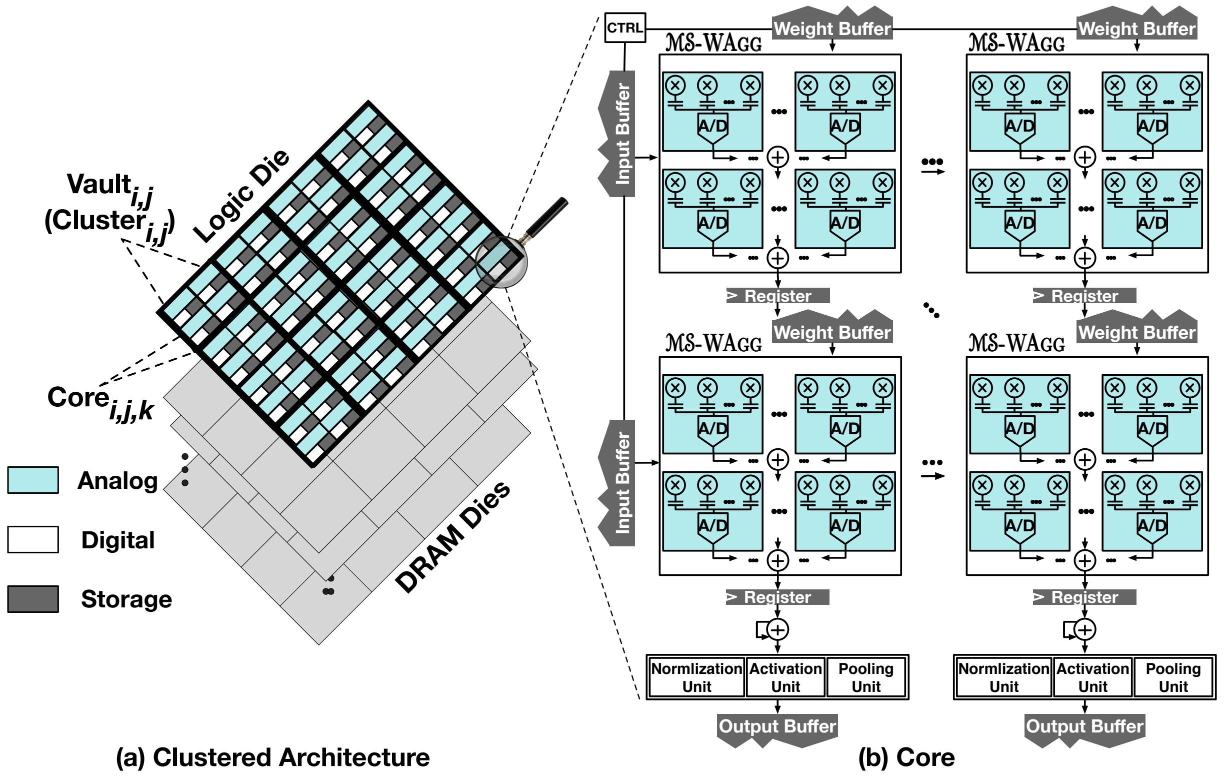


Figure 2.8. Hierarchical clustered architecture

2.4.3 Hierarchically Clustered Architecture

As illustrated in Figure 5.10, a collection of MS-WAGGs constitute an accelerator core from which the clustered architecture of BIHIWE is designed. The three aforementioned optimization techniques, results in 5.4× less energy for a single 8-bit MACC in comparison with a digital logic. Hence, it is possible to integrate a larger number of mixed-signal compute units in a given power budget compared to a digital architecture. To efficiently utilize this increase in compute units, a high bandwidth memory substrate is required. To maximize the benefits of the mixed-signal computation, 3D-stacked memory is an attractive option since it reduces the energy cost of data accesses and provides a higher bandwidth for data transfer between the on-chip compute and off-chip memory [92, 135]. Based on these insights, we devise a clustered architecture for BIHIWE with a 3D-stacked memory substrate as shown in Figure 5.10. As the results in Section 5.6.2 Figure 2.16 shows, a flat design would result in significant

underutilization of the compute resources and bandwidth from 3D stacking. Therefore, BiHiwe is a hierarchically clustered architecture that allocates multiple accelerator cores as a cluster to each vault (Figure 5.10(a)). Figure 5.10(b) depicts a single core. As shown in Figure 5.10(b), each core is self-sufficient and packs a mixed-signal systolic array of $\mathcal{MS}$-WAggs as well as the digital Pooling Unit, Activation Unit, and Normalization Unit, etc. The mixed-signal array is responsible for the convolutional and fully connected layers. Generally, wide and interleaved bit-partitioned execution within $\mathcal{MS}$-WAggs is orthogonal to the organization of the accelerator architecture. This work explores how to embed them and the proposed compute model, within a systolic design and enables end-to-end programmable mixed-signal acceleration for a variety of DNNs.

Accelerator core. As Figure 5.10(b) depicts, the first level of hierarchy is the accelerator core and its 2D systolic array that utilizes the $\mathcal{MS}$-WAggs. As depicted, the Input Buffers and Output Buffers are shared across the columns and rows, respectively. Each $\mathcal{MS}$-WAgg has its own Weight Buffer. This organization is commensurate with other designs and reduces the cost of on-chip data accesses as inputs are reused with multiple filters [128]. However, what makes our design different is the fact that each buffer needs to supply a sub-vector not a scalar in each cycle to $\mathcal{MS}$-WAggs. The rewiring of the inputs and weights is already done inside the $\mathcal{MS}$-WAggs since the size of bit-partitions is fixed. Consequently, there is no need to reformat any of inputs, activations, or weights. To preserve the accuracy of the DNNs, intermediate results are stored as 32-bit digital values and intra-column aggregations are performed in digital mode.

On-chip data delivery for accelerator cores. To minimize data movement and exploit the abundant data-reuse in DNNs, BiHiwe uses a statically-scheduled interconnect that is capable of multicasting/broadcasting data across accelerator cores. Static scheduling enables the BiHiwe compiler stack to do exhaustive search over variegated possibilities of cutting and tiling DNN layers across cores to maximizing inter- and intra-core data-reuse. The static schedule is encoded in the form of data communication instructions.

Parallelizing computations across accelerator cores. To minimize data movement, the BiHiwe clustered architecture (1) divides the computations into tiles that fit within the on-chip capacity of the scratchpads, and (2) *cuts* the tiles of computations across cores to minimize DRAM accesses by maximally utilizing the multicast/broadcast capabilities of BiHiwe on-chip data delivery network. To simplify the hardware, scratchpad buffers are private to each core and the shared data is replicated across multiple cores. Thus, a single *tile* of data can be read once from the memory and then be broadcasted/multicasted across cores to reduce DRAM accesses. The cores use double-buffering to hide the latency for memory accesses for subsequent tiles. Cores use *output-stationary* dataflow that minimizes the number of A/D conversions by accumulating results in the charge-domain. Section 4.5 discusses the cutting/tiling optimizations in compiler.

2.4.4 BiHiwe Instruction Set

The BiHiwe ISA provides a layer of abstraction that exposes the following unique properties of its architecture to the compiler (1) mixed-signal execution within a BiHiwe core; and (2) data-movement for both 3D-stacked memory and on-chip software-managed scratchpads between different BiHiwe cores. As such, BiHiwe uses a block-structured ISA where the blocks have repetition counters due to the tile-based execution and segregates the execution of the DNN into (1) data communication instruction blocks that transfer tiles of data between 3D-stacked memory and on-chip scratchpads (Input Buffer/Weight Buffer/Output Buffer in Figure 5.10) using address generation instructions, and (2) compute instruction blocks that consumes the tile of data from a communication instruction block to produce an output tile. The communication block and compute block together specify a static schedule for DNN execution in BiHiwe.

Using the compute instruction block, the compiler has complete control over on-chip scratchpads, A/D conversion rate, and bit-partitioning across $\mathcal{MS}$-WAggs. These pieces of information are encoded in the header of the compute instruction blocks. The granularity of bit-partitioning and charge-based accumulation is determined for each microarchitectural implementation based on technology node and circuit design style. As such, to support different

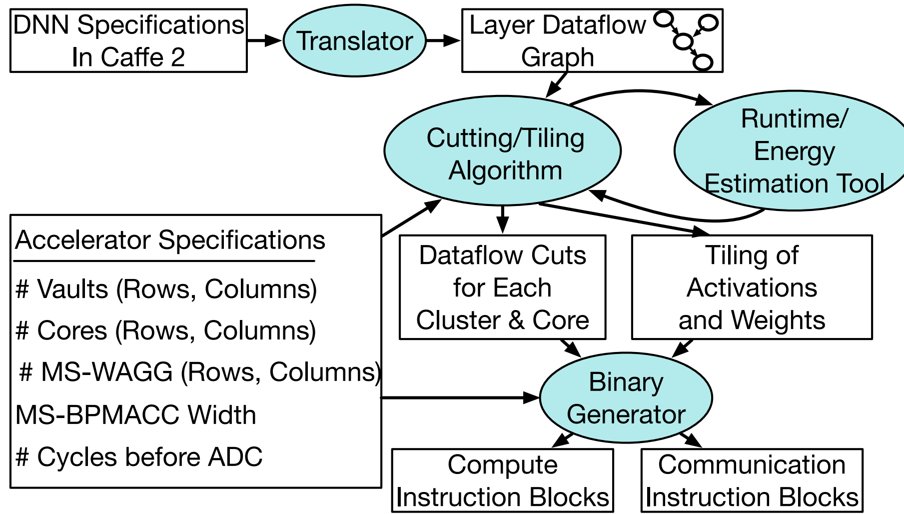


Figure 2.9. BiHiwe compilation stack.

technology nodes and designs and allow extensions to the architecture, the BIHIWE ISA encodes the bit-partitioning and accumulation cycles. Using the communication instruction blocks, the compiler stack exploits the broadcasting/multicasting capabilities to optimize data movement while maximizing data locality for the on-chip scratchpad memories in each core.

2.5 BIHIWE Compiler Stack

Figure 2.9 illustrates the BIHIWE compiler stack that accepts a high-level Caffe2 [87] specification of the DNN to generate an instruction binary (BIHIWE ISA). The first step in the compiler stack is a translation of the Caffe2 file into a layer DataFlow Graph (DFG) that preserves the structure of the DNN. The BIHIWE compiler stack also accepts a specification of the accelerator configuration that includes the organizations and configurations (# rows, #columns) of the clusters, vaults, and cores as well as details of the $\mathcal{MS}$-BPMACCs. Using the layer DFG and the accelerator configuration, the compiler then proceeds with an optimization step that determines the optimal cut of the DFG nodes across BIHIWE clusters and cores, and optimal tile sizes for the multidimensional arrays (DFG edges) to fit into the limited on-chip memory. For each node in the layer DFG, the optimization algorithm performs an exhaustive search of different cuts and tile sizes for incoming and outgoing edges. For each candidate

```
Initialize cut_opt[N] ← ∅;    Initialize tiling_opt[N] ← ∅
for layer_i ∈ DFG_DNN do
    s_opt ← ∞
    for tiling_{i,j} ∈ layer_i do
        for cut_{i,j,k} ∈ tiling_{i,j} do
            (runtime_{i,j,k}, energy_{i,j,k}) ← EstimTool(tiling_{i,j}, cut_{i,j,k})
             s_{i,j,k} ← runtime_{i,j,k} × energy_{i,j,k}
            if s_{i,j,k} < s_opt then
                cut_opt[i] ← cut_{i,j,k};    tiling_opt[i] ← tiling_{i,j}
return cut_opt, tiling_opt
```

Algorithm 1: Cutting/tiling algorithm for clustered acceleration.

cut and tile-size, the compiler stack uses an analytical estimation tool that determines the total energy consumption and runtime. Estimation is viable, as the DFG does not change, there is no hardware managed cache, and the accelerator architecture is fixed during execution. Thus, there are no irregularities that can hinder estimation. Algorithm 1 depicts the cutting/tiling procedure. When cuts and tiles are determined, the compiler generates the binary code that contains the communication and computation instruction blocks in BIHIWE ISA.

2.6 Mitigating Analog Non-Idealities

Although analog circuitry offers significant reduction in energy, they might lead to accuracy degradation. Thus, their error needs to be properly modeled and accounted for. Specifically, $\mathcal{MS}$-BPMACCS, the main analog component, can be susceptible to (1) thermal noise, (2) computational error caused by incomplete charge transfer, and (3) PVT variations. Traditionally, analog circuit designers mitigate sources of error by just configuring hardware parameters to values which are robust to non-idealities. Such hardware parameter adjustments require rather significant energy/area overheads that scale linearly with number of modules. However, due to the scaled-up nature of our design, we need to mitigate these non-idealities in a higher and algorithmic level. We leverage the training algorithm's inherent mechanism to reduce error (loss)

and use mathematical models to represent these non-idealities. We, then, apply these models during forward pass to *adjust and fine-tune pre-trained neural models with just a few more epochs* across the chips within a technology node. Our approach is commensurate with recent work [200] that uses fine-tuning passes to incorporate analog non-idealities. The rest of this section details non-idealities and their modeling.

Thermal noise. Thermal noise is an inherent perturbation in analog circuits caused by the thermal agitation of electrons. This noise can be modeled according to a normal distribution, where the ideal voltage deviates relative to a value comprised of the working temperature (T), Boltzmann constant (k), and capacitor size (C) which produce the deviation $\sigma = \sqrt{kT/C}$. Within BIHIWE, switched-capacitor MACC units are mainly effected by the combined thermal noise resulting from weights and accumulator capacitors (C_W and C_{ACC} respectively). The noise from these capacitors gets accumulated during the m cycles of computation for each individual MACC unit and then gets aggregated across the n MACC units in $\mathcal{MS}$-BPMACC. By applying the thermal noise equation used for similar MACC units [151] to a $\mathcal{MS}$-BPMACC unit, the standard deviation at the output is described by Equation 2.5:

$$\sigma_{ACC} = \sqrt{\frac{kT(\alpha|W_{m-1}|+3\alpha+3)}{9\alpha(\alpha+1)^2C_w}\left(\sum_{i=0}^{m-1}\left(\frac{\alpha}{1+\alpha}\right)^{2i}\right)\times n} \tag{2.5}$$

In the above equation, α is equal to $\frac{C_{ACC}}{3C_W}$. We add error tensors to outputs of convolutional/fully connected layers in DNN forward propagation, to incorporate thermal noise effect. Elements of error tensors are sampled from a normal distribution as $\mathcal{N}(\mu = 0, \sigma^2 = (\sigma_{ACC} \times r \times 85)^2)$. σ_{ACC} is scaled by r, the amount of $\mathcal{MS}$-BPMACC operations required to generate an element in the output feature map, as well as the amount of total bit-shifts applied to each result by $\mathcal{MS}$-WAGG engine, 85.

Computational error. Another source of error in BIHIWE's computations arises when charge is shared between capacitors during the multiplication and accumulation. Within each MACC unit, the input capacitors (C_X) transfer a sampled charge to the weight capacitors (C_W) to produce

charge proportional to the multiplication result. But the resulting charge is subject to error dependent on the ratio of weight and input capacitor sizes ($\beta = C_1/C_2$) as shown in Equation 2.3. This shared charge in the weight capacitors introduces more error when it is redistributed to the accumulating capacitor (C_{ACC}) which cannot absorb all of the charge, leaving a small portion remaining on the weight capacitors in subsequent cycles. The ideal voltage ($V_{ACC,Ideal}$) produced after m cycles of multiplication can be derived from Equation 2.4 as follows:

$$V_{ACC,Ideal}[m] = \sum_{i=1}^{m} \frac{V_{\text{DD}}}{9\alpha} W_i X_i \tag{2.6}$$

By considering the computational error from incomplete charge sharing, the actual voltage at the accumulating capacitor after m cycles of MACC operations ($V_{ACC,R}[m]$) becomes:

$$\frac{3\alpha}{3\alpha+|W_m|} V_{ACC,R}[m-1] + \frac{W_m X_m \beta}{(3\alpha+|W_m|)(3\beta+|W_m|)} V_{\text{DD}} \tag{2.7}$$

We consider computational error in the fine-tuning pass by including the multiplicative factors shown in Equation 2.7 in weights. During the forward pass, the fine-tuning algorithm decomposes weight tensors in convolutional/fully-connected layers into groups corresponding to $\mathcal{MS}$-WAGG configuration and updates the individual weight values (W_i) to new values (W_i') with the computational error:

$$W_i' = \frac{W_i}{3\alpha + |W_i|} \frac{\beta V_{DD}}{3\beta + |W_i|} \prod_{j=i+1}^{m-1} \frac{3\alpha}{3\alpha + |W_j|} \qquad \forall 0 \le i \le m-1 \tag{2.8}$$

Process variations. We use the sizing of the capacitors to provision and mitigate for the process variations to which the switched-capacitor circuits are generally robust. This is effective because the capacitors are implemented using a number of smaller unit capacitors with common-centroid layout technique [120]. We, specifically, use the metal-fringe capacitors for MACCs with mismatch of just 1% standard deviation [252] with the max variation of 6% (6σ) which is well

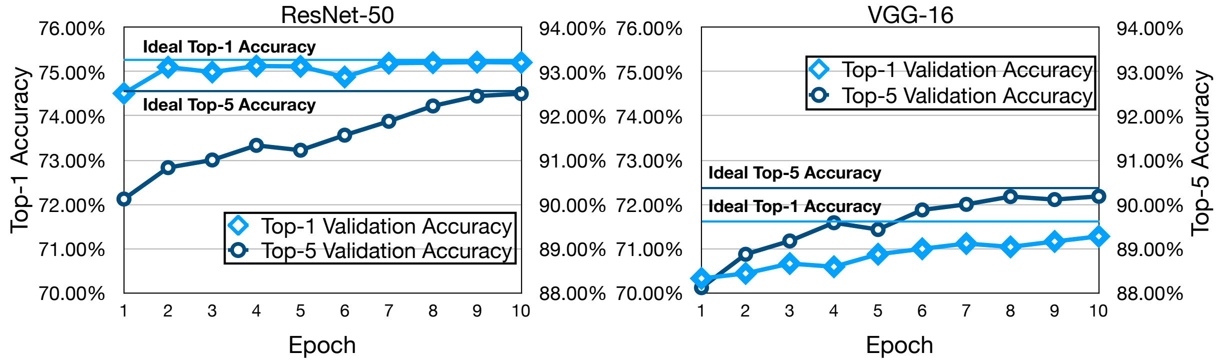


Figure 2.10. ResNet-50 and VGG-16 accuracy after fine-tuning.

below the error margins considered for the computational error.

Temperature variations. This is modeled by adding a perturbation term to T in Equation 2.5 as a gaussian distribution $\mathcal{N}_T(\mu, \sigma^2)$. We consider the maximum value of the temperature as 358 commensurate with existing practices [81], and the minimum value as 300(This is the peak-to-peak range for the gaussian distribution (6σ)).

Voltage variations. We also model the voltage variation by adding a gaussian distribution to V_{DD} term in Equation 2.8. Our experiments show that, variations in voltage can be mitigated up to 20%. The extensive amount of vector dot-product operations in DNNs, allows for the minimum and maximum values of the distributions being sampled sufficient amount of times, leading to coverage of the corner cases.

Atop all these considerations, we use differential signaling for ADCs which attenuates the common-mode fluctuations such as PVT variations. To show the effectiveness of our techniques, Figure 2.10 plots the result of fine-tuning process of two benchmarks, ResNet-50 and VGG-16 for ten epochs. Table 2.3 reports the summary of accuracy trends for all the benchmarks, which achieve less than 0.5% loss. As Figure 2.10 shows, the fine-tuning pass compensates the initial loss (0.73% for top-1 and 2.41% for top-5) to only 0.04% for top-1 and 0.02% for top-5. VGG-16 is slightly different and reduces the initial loss (1.16% for top-1 and 2.24% for top-5) to less than 0.18% for top-1 and 0.13% for top-5 validation accuracy. The trends are similar for other benchmarks and omitted due to space constraints.

Table 2.1. Evaluated benchmark DNNs

DNN	Type	Domain	Dataset	Multiply-Adds	Model Weights
AlexNet [141]	CNN	Image Classification	Imagenet [75]	2,678 MOps	56.1 MBytes
CIFAR-10 [224, 117]	CNN	Image Classification	CIFAR-10 [142]	617 MOps	13.4 MBytes
GoogLeNet [241]	CNN	Image Classification	Imagenet	1,502 MOps	13.5 MBytes
ResNet-18 [110]	CNN	Image Classification	Imagenet	4,269 MOps	11.1 MBytes
ResNet-50 [110]	CNN	Image Classification	Imagenet	8,030 MOps	24.4 MBytes
VGG-16 [224]	CNN	Image Classification	Imagenet	31 GOps	131.6 MBytes
VGG-19 [224]	CNN	Image Classification	Imagenet	39 GOps	137.3 MBytes
YOLOv3 [199]	CNN	Object Recognition	Imagenet	19 GOps	39.8 MBytes
PTB-RNN [117]	RNN	Language Modeling	Penn TreeBank [170]	17 MOps	16 MBytes
PTB-LSTM [115]	RNN	Language Modeling	Penn TreeBank	13 MOps	12.3 MBytes

Table 2.2. BiHiwe and baselines platforms

Parameters	ASIC		Parameters	GPU	
Chip	**BiHiwe**	**Tetris**	**Chip**	**RTX 2080 TI**	**Titan Xp**
MACCs	16,384	3,136	**Tensor Cores**	544	—
On-chip Memory	9216 KB	3698 KB	**Memory**	11 GB (GDDR6)	12 GB (GDDR5X)
Chip Area (mm^2)	122.3	56	**Chip Area** (mm^2)	754	471
			Total Dissipation Power	250 W	250 W
Frequency	500 Mhz	500 Mhz	**Frequency**	1545 Mhz	1531 Mhz
Technology	45 nm	45 nm	**Technology**	12 nm	16 nm

2.7 Evaluation

2.7.1 Methodology

Benchmarks. We use ten diverse CNN/RNN models including real-time object recognition and word-level language modeling, described in Table 3.1. These benchmarks includes medium to large scale models and variety of multiply-add operations.

Simulation infrastructure. We develop a cycle-accurate simulator and a compiler for BIHIWE. The simulator dumps the statistics of runtime and accesses to all components and calculates the power. Since, all the instructions are statically scheduled, the simulator can calculate the exact number of accesses to components.

Iso-power and iso-area comparison with TETRIS. We match the on-chip power of BIHIWE and TETRIS and compare the total runtime and energy, including DRAM accesses. TETRIS supports 16-bit execution while BIHIWE supports 8-bit. For fairness, we modify the open-

source TETRIS simulator [93] and proportionally scale its runtime/energy. BIHIWE supports 8-bit since this representation has virtually no impact by itself on the accuracy of the DNNs [117, 298, 176, 154, 292].

Comparison with analog/digital accelerators. We also compare BIHIWE to mixed-signal RedEye [158], two analog memristive accelerators [215, 228], and Google TPU [128], all in 8-bits. The original designs [215, 228] use 16-bits. We optimistically increase the efficiency of the competitor designs by 4× to model 8-bit execution.

GPU comparison. We also compare BIHIWE to two Nvidia GPUs, RTX 2080 TI with tensor cores and Titan Xp (Table 3.2). For a fair comparison, we use 8-bit on GPUs using Nvidia's TensorRT 5.1 [18] library compiled with the optimized cuDNN 7.5 and CUDA 10.1.

Energy and area measurement. All hardware modelings are performed using FreePDK 45-nm standard cell library [10]. We implement the switched-capacitor MACCs in Cadence Analog Design Environment V6.1.3 and use Spectre SPICE V6.1.3 to model the system. We then, use Layout XL of Cadence to extract the energy/area. The energy/area for ADCs are obtained from [179]. We implement digital blocks of BIHIWE, including adders, shifters, and interconnection in Verilog RTL and use Synopsys Design Compiler (L-2016.03-SP5) for synthesis and measuring energy/area. We use CACTI-P [155] to model on-chip buffers. 3D-stacked DRAM is based on HMC [65, 122], same as TETRIS, and the bandwidth and access energy are adopted form that work.

Error modeling. We use Spectre SPICE V6.1.3 to extract noise behavior of MACCs. Thermal noise, computational error, and PVT variations are considered based on details in Section 2.6. We implement extracted hardware error models and corresponding mathematical modelings using PyTorch v1.0.1 and integrate them into Neural Network Distiller v0.3 framework [303] for a fine-tuning pass over evaluated benchmarks.

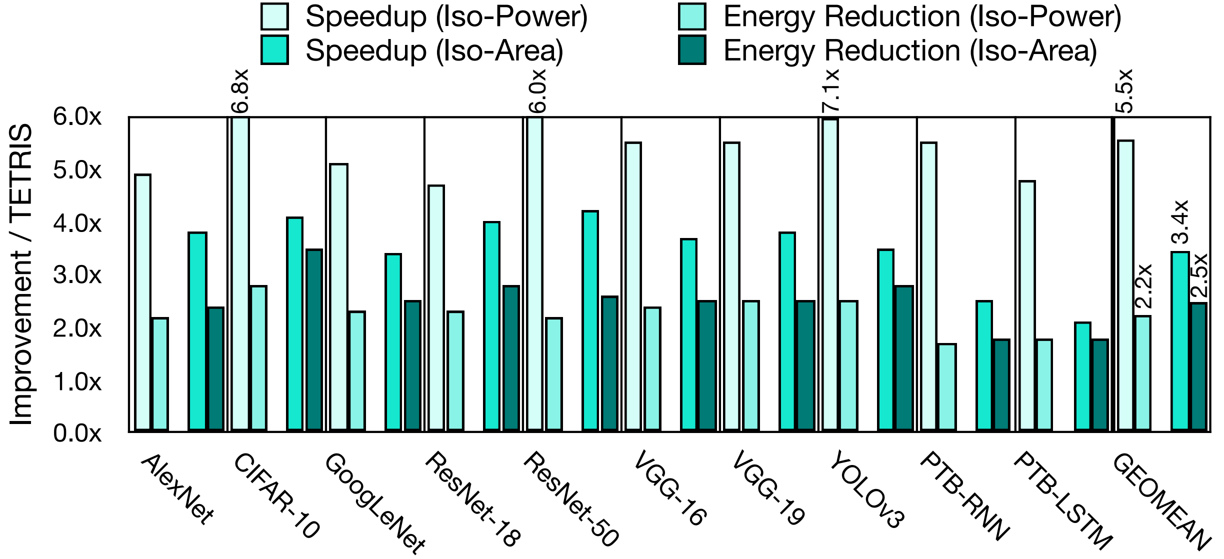


Figure 2.11. Iso-Power/Iso-Area speedup and energy reduction over Tetris.

2.7.2 Experimental Results

Comparison with TETRIS

Iso-power and iso-area comparisons. Figure 2.11 shows the performance and energy reduction of BIHIWE over TETRIS. On average, BIHIWE delivers a 5.5×speedup over TETRIS in iso-power setting. The low power and wide bit-partitioned mixed-signal design of $\mathcal{MS}$-WAGGs in BIHIWE enables us to integrate 5.2× more compute units than TETRIS in the same power budget. The highest speedup is observed in YOLOv3 and CIFAR-10, where the network topology favors the wide vectorized execution in BIHIWE. The lowest speedup is observed in ResNet-18, since its relatively small size leads to under-utilization of compute resources in BIHIWE. Figure 2.11 also demonstrates total energy reduction for BIHIWE as compared to TETRIS in iso-power setting. On average, BIHIWE yields 2.2×energy reduction. The lowest energy reduction is observed in RNN benchmarks, PTB-RNN and PTB-LSTM, since matrix-vector operations in RNNs require significant number of DRAM accesses for weights, limiting benefits.

Figure 2.11 also shows iso-area comparisons. Scaling-up computes in TETRIS by 2.25× to match the area of BIHIWE results in ≈ 60% increase in TETRIS performance. This improvement in performance comes at a cost of reduced energy-efficiency due to an increase in memory

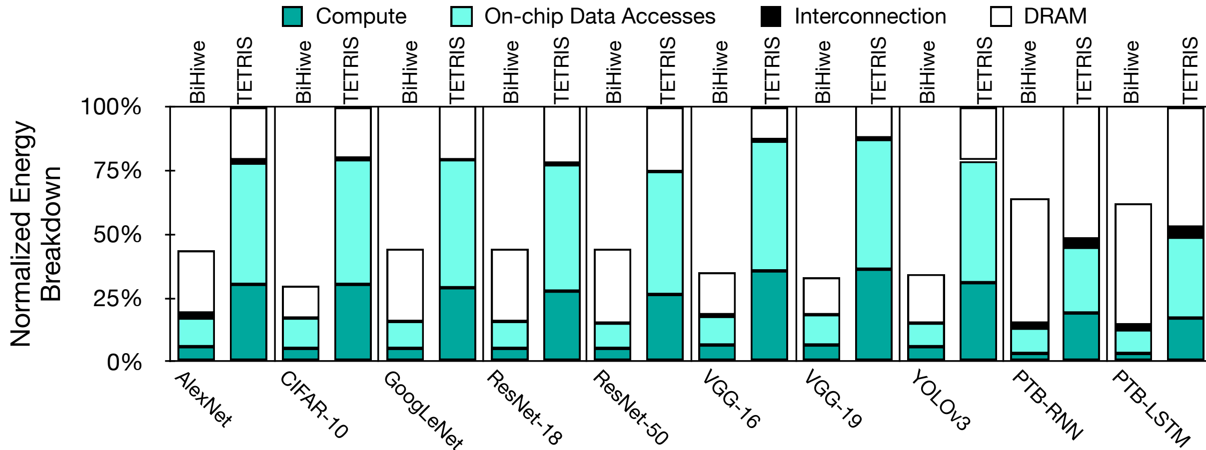


Figure 2.12. Energy breakdown of BiHiwe and Tetris.

accesses to feed the additional compute units. Trends in speedup and energy-reduction remain the same with the exception of ResNet-18, which now sees resource underutilization in TETRIS. Overall, BIHIWE shows 3.4×speedup and 2.5×energy reduction.

Energy breakdown. Figure 2.12 shows the energy breakdown normalized to TETRIS across: (1) on-chip compute units, (2) on-chip memory, (3) interconnect, and (4) 3D-stacked DRAM. DRAM accesses account for the highest portion of the energy in BIHIWE, since BIHIWE significantly reduces the on-chip compute energy. While BIHIWE has a larger number of compute resources compared to TETRIS, the number of DRAM accesses remain almost the same. This is because the statically-scheduled interconnect allows data to be multicasted/broadcasted across multiple cores in BIHIWE without significantly increasing the number of DRAM accesses. Unlike the fully-digital PEs in TETRIS, BIHIWE uses $\mathcal{MS}$-WAGGs which perform wide vectorized operations. Each MACC operation in BIHIWE consumes 5.4× less energy compared to TETRIS. The output-stationary dataflow enabled by capacitive accumulation in addition to the systolic organization of $\mathcal{MS}$-WAGGs in each core of BIHIWE eliminates the need for register file, leads to 4.4× reduction in on-chip data movement.

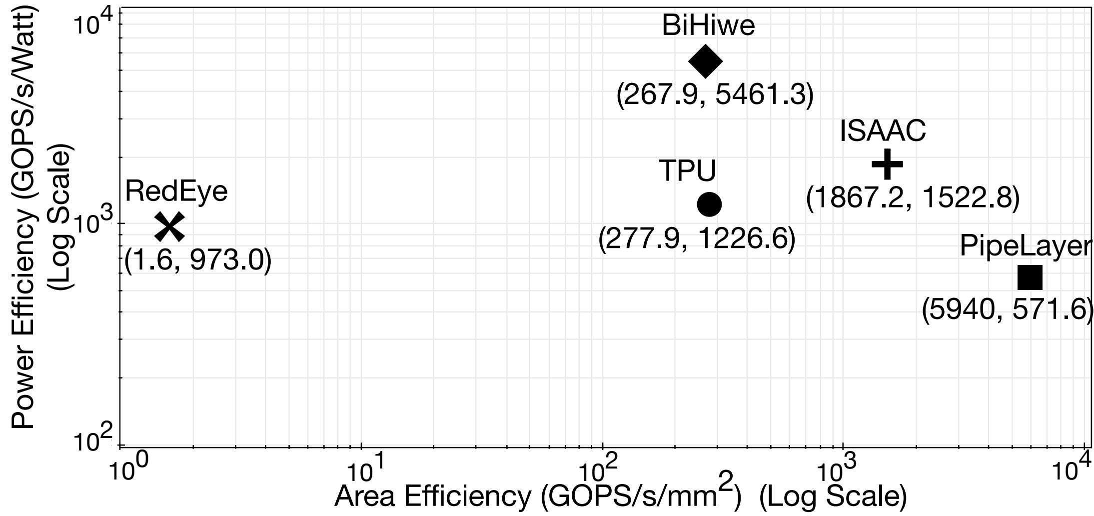


Figure 2.13. Comparison with other accelerators.

Comparison with Other Baselines

Figure 2.13 depicts power efficiency (GOPS/s/Watt) and area efficiency (GOPS/s/mm^2) of BIHIWE with other recent accelerators. Due to the lack of available raw performance/energy numbers for specific DNNs and the fact that the simulation/compilation infrastructures for prior accelerators are not open sourced, we use these metrics that is commensurate with comparisons for recent designs [30, 228, 164, 32] to provide a best effort analysis. On average for the evaluated benchmarks, BIHIWE achieves 81% of its peak efficiency.

Mixed-signal CMOS: RedEye [158]. RedEye also uses switched-capacitor circuitry. Compared to RedEye, BIHIWE offers 5.5× power efficiency and 167× area efficiency. In contrast to RedEye [158] which does not exploit any sort of bit-partitioning, the proposed wide, interleaved, and bit-partitioned arithmetic amortizes the cost of ADCs in BIHIWE and yields these benefits.

Analog Memristive designs [215, 228]. Prior work in ISAAC and PipeLayer have explored memristive technology for DNN acceleration, which integrates both compute and storage in the same die, offering higher compute density compared to traditional CMOS. Generally, memrisitive designs perform computations in current domain, requiring costly ADCs to sample currents at high rates, curtailing the power-efficiency. Overall, compared to ISAAC and

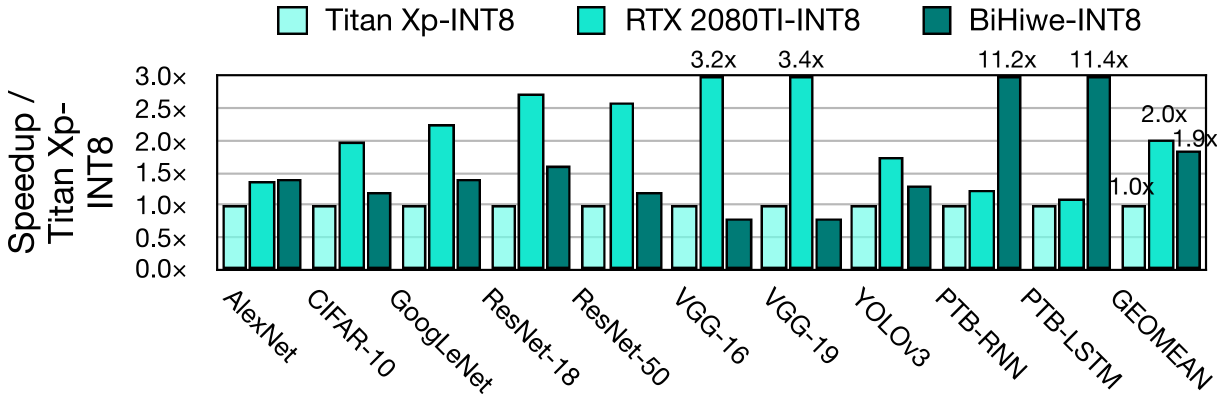


Figure 2.14. Performance comparison to GPUs.

PipeLayer, BiHiwe improves the power efficiency by 3.6× and 9.6×, respectively.

Google TPU [128]. Compared to TPU, which also uses systolic design, BiHiwe delivers 4.5× more peak power efficiency and almost the same area efficiency. Leveraging the wide, interleaved, and bit-partitioned arithmetic with its switched-capacitor design in BiHiwe, reduces the cost of MACC operations significantly.

Comparison with GPUs. Figure 2.14 compares performance of BiHiwe with Titan Xp and RTX 2080 TI, normalized to Titan Xp. BiHiwe, on average, yields 1.9× speedup over Titan Xp and is just 5% slower than RTX 2080 TI. CNNs require abundant matrix-matrix multiplications, well-suited for tensor cores, leading to RTX 2080 TI's outperformance on both BiHiwe and Titan Xp. However, BiHiwe outperforms RTX 2080 TI in PTB-RNN and PTB-LSTM with 11.2× and 11.4×, respectively. RNNs require matrix-vector multiplications– particularly suitable for the wide vectorized operations supported in $\mathcal{MS}$-WAGGs. However, BiHiwe outperforms both Titan Xp and RTX 2080 TI GPUs in Performance-per-Watt by large margins of 70.1×and 35.4×, respectively.

Design Space Explorations

Design space exploration for bit-partitioning. Figure 2.15 shows the reduction in energy and area for different bit-partitioning design points that are algorithmically identical and perform the same 8-bit×8-bit vector dot-product with 32 elements. However, the baseline design uses 8-

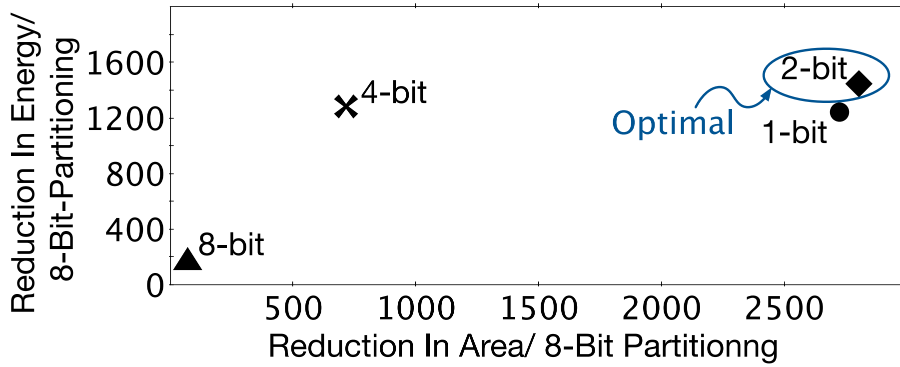


Figure 2.15. Design space exploration for bit-partitioning.

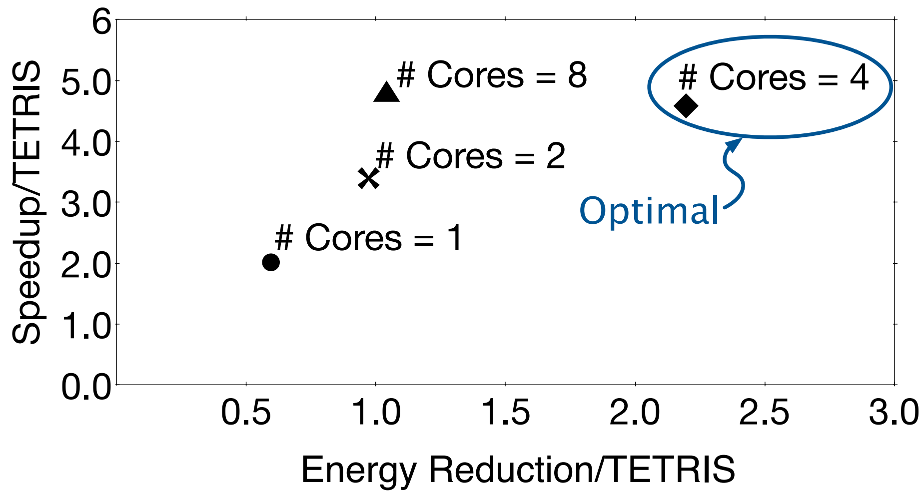


Figure 2.16. Design space exploration for # core per cluster.

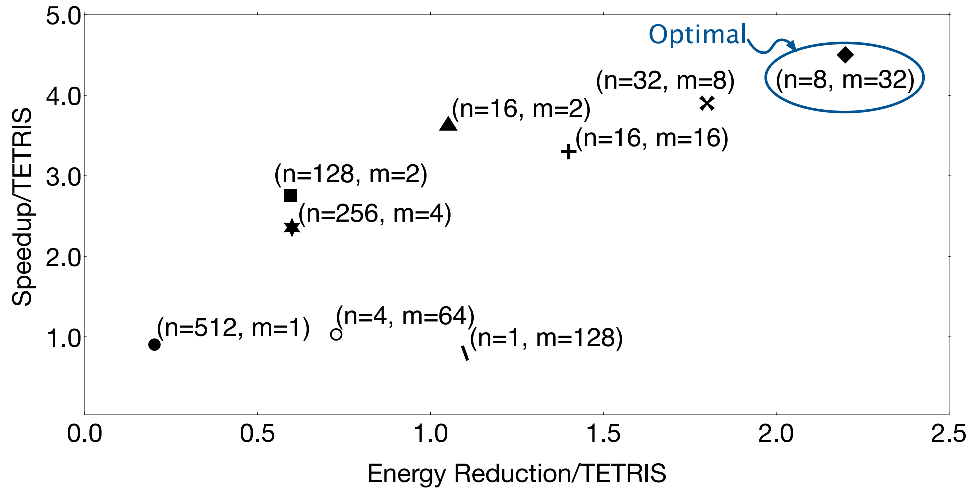


Figure 2.17. Design space exploration for $\mathcal{MS}$-BPMacc.

bit×8-bit MACC units while the rest use our wide and interleaved bit-partitioned arithmetic. As depicted, 2-bit partitioning strikes the best balance in energy/area with switched-capacitor design of MACC units at 45 nm. Compared to 2-bit, single-bit partitioning quadratically increases the number of low bitwidth MACCs from 16 (2-bit partitioning) to 64 (1-bit partitioning) to support 8-bit operations. This imposes disproportionate overhead that outweighs benefit of decreasing MACC units energy/area.

Design space exploration for clustered architecture. BiHiwe uses a hierarchical architecture with multiple cores in each vault. Having a larger number of small cores for each vault yields increased utilization of compute resources, but requires data transfer across cores and replication. We explore the design space with 1, 2, 4, and 8 cores per cluster. As Figure2.16 shows, BiHiwe with four cores per each vault (default configuration) is optimal by striking a better balance between data accesses and compute resource utilization. Configuration with 8-cores results in higher data accesses, hence higher energy.

Design space exploration for $\mathcal{MS}$-BPMacc configuration. The number of accumulation cycles (m) before A/D conversion and the number of MACC units (n) are two main parameters of $\mathcal{MS}$-BPMacc which define ADC resolution and sample rate, determining its power. Figure 2.17 shows the design space exploration for different configurations of $\mathcal{MS}$-BPMacc. In a fixed

Table 2.3. Accuracy before and after fine-tuning.

DNN Model	Dataset	Top-1 Accuracy (With Non-Idealities)	Top-1 Accuracy (After Fine-Tuning)	Top-1 Accuracy (Ideal)	Final Accuracy Loss
AlexNet	Imagenet	53.12%	56.64%	57.11%	0.47 %
CIFAR-10	CIFAR-10	90.82%	91.01%	91.03%	0.02 %
GoogLeNet	Imagenet	67.15%	68.39%	68.72%	0.33 %
ResNet-18	Imagenet	66.91%	68.96%	68.98%	0.02 %
ResNet-50	Imagenet	74.5%	75.21%	75.25%	0.04 %
VGG-16	Imagenet	70.31%	71.28%	71.46%	0.18 %
VGG-19	Imagenet	73.24%	74.20%	74.52%	0.32 %
YOLOv3	Imagenet	75.92%	77.1%	77.22%	0.21 %
PTB-RNN	Penn TreeBank	1.1 BPC	1.6 BPC	1.1 BPC	0.0 BPC
PTB-LSTM	Penn TreeBank	97 PPW	170 PPW	97 PPW	0.0 PPW

power budget for compute units, we measure total runtime/energy of BIHIWE across benchmarks and normalize it to those of TETRIS. As shown in Figure 2.17, increasing number of MACCs, limits the number of accumulation cycles and results in high sample rate ADCs. Using high sample-rate ADCs significantly increases power. On the other hand, increasing number of accumulation cycles, limits the number of MACCs, which restricts the number of $\mathcal{MS}$-WAGGs that can be integrated under given power budget. Overall, the optimal design point that delivers the best performance and energy constitutes eight MACC units and 32 accumulation cycles.

Evaluation of Circuitry Non-Idealities

Table 2.3 shows the Top-1 accuracy With Non-Idealities, After Fine-Tuning, Ideal, and the Final Accuracy Loss. As shown in Table 2.3 AlexNet and ResNet-18 are more sensitive to the non-idealities, leading to a higher initial accuracy degradation. To recover the accuracy loss, we perform a fine-tuning step for a few epochs. By performing this fine-tuning step, the accuracy loss of the CIFAR-10, ResNet-18, and ResNet-50 networks is fully recovered (loss is less than 0.04%) which within these networks, CIFAR-10 and ResNet-50 are more robust to non-idealities. Accuracy loss for other networks is below 0.5% which within those AlexNet has maximum loss. Both PTB-RNN and PTB-LSTM recover all the loss after fine-tuning. The final results after fine-tuning step show the effectiveness of this approach in recovering the accuracy loss due to the non-idealities pertinent to analog computation.

2.8 Related Work

There is a large body of work on digital DNN accelerators [86, 58, 92, 73, 168, 295, 29, 129, 221, 64, 184, 30, 106, 55, 56, 135, 128, 54, 222, 27, 112, 152, 279, 211, 203, 118, 206, 98]. Mixed-signal acceleration has also been explored previously for neural networks [253, 234] and is gaining traction for deep models [215, 233, 158, 37, 150, 39, 44, 294, 151]. This paper fundamentally differs from these inspiring efforts as it delves into mathematics of DNN operations, reformulates and defines the interleaved and bit-partitioned arithmetic combined with charge-domain computation to overcome challenges in mixed-signal acceleration. Below, we discuss the most related works.

Switched-capacitor design. Switched-capacitor circuits [101] have a long history, having been mainly used for designing amplifiers [67], ADC/DAC [89] and filters [43]. They have been used even for the previous generation of neural networks [253]. More recently, they have also been used for matrix multiplication [38, 151], which can benefit DNNs. This work takes inspiration from these efforts but differs from them in that it defines and leverages wide, interleaved, and bit-partitioned reformulation of DNN operations. Additionally, it offers a comprehensive architecture to accelerate a wide variety of DNNs.

Programmable mixed-signal accelerators. PROMISE [233] offers a mixed-signal architecture that integrates analog units within the SRAM memory blocks. RedEye[158] is a low-power near-sensor mixed-signal accelerator that uses charge-domain computations. These works do not offer wide interleavings of bit-partitioned basic operations as described in this paper.

Fixed-functional mixed-signal accelerators. They are designed for a specific DNN. Some focus on handwritten digit classification [38, 177] or binarized mixed-signal acceleration of CIFAR-10 images [39]. Another work focuses on spiking neural networks' acceleration [44]. In contrast, our design is programmable and supports interleaved bit-partitioning.

Resistive memory accelerators. There is a large body of work using resistive memory [215, 59, 228, 164, 32, 190, 124, 153, 48, 125, 277]. We provided direct comparison to ISAAC [215] and

PipeLayer [228]. ISAAC most notably introduces the concept of temporally bit-serial operations, also explored in PRIME [59], and is augmented with spike-based data scheme in PipeLayer. BIHIWE, in contrast, formulates a partitioning that spatially regroups lower-bitwidth MACCs across different vector elements and performs them in-parallel. PRIME does not provide absolute measurements and its simulated baseline is not available for head-to-head comparisons. PRIME also uses multiple truncations that change the mathematics. Conversely, our formulation does not induce truncation or mathematical changes.

Bit-level composable designs. Bit Fusion [222] proposes bit-level dynamic composability to support quantized DNNs. BitBlade [206] and BPVeC [98] extends bit-level reconfigurability to vector-level composability to amortize the energy and area cost of bit-flexibility. In contrast, this work delves into the details of mixed-signal computing, proposes wide, interleaved, and bit-partitioned arithmetic, and combines it with switched-capacitor circuits to enable mixed-signal acceleration.

2.9 Conclusion

This work proposed wide, interleaved, and spatially bit-partitioned arithmetic to overcome key challenges in mixed-signal acceleration of DNNs. This arithmetic enabled rearranging the highly parallel MACC operations in DNNs into wide low-bitwidth efficient mixed-signal computations. Further, we use switched-capacitor circuitry that reduces the rate of ADC by accumulating partial results in the charge domain. The incarnate design, BIHIWE, offers significant benefits over its state-of-the-art analog and digital counterparts.

2.10 Acknowledgement

Chapter 2 is a partial reprint of the material as it appears in: S. Ghodrati, H. Sharma, S. Kinzer, A. Yazdanbakhsh, J. Park, N. Kim, D. Burger, and H. Esmaeilzadeh, “Mixed-Signal Charge-Domain Acceleration of Deep Neural Networks through Interleaved Bit-Partitioned

Arithmetic." in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2020. The dissertation author was the primary investigator and author of this paper.

Chapter 3

Interweaving Data-Level and Bit-Level Parallelism for Energy-Efficient Digital Acceleration

3.1 Introduction

The growing body of neural accelerators [55, 29, 129, 106, 215, 59, 184, 128, 222, 152, 90] exploit various forms of Data-Level Parallelism (DLP) that are abundant in Deep Neural Networks. For instance, Google's TPU [128] extracts data-level parallelism across the lanes of its systolic array, Microsoft's Brainwave [90] builds a dataflow architecture from vectorized units, and GPUs have been long designed for Single-Instruction Multiple-Data (SIMD) execution model. Nonetheless, these various organization still rely on isolated self-sufficient units that process all the bits of input operands and generate all the bits of the results. These values, packed as atomic words, are then communicated through an operand delivery-aggregation interconnect for further computation. This work sets out to explore a different design where each unit in vectorized engines is only responsible for processing a bit-slice. This design offers an opening to explore the interleaving of Bit-Level Parallelism with DLP for neural acceleration. Such an interleaving opens a new tradeoff space where the complexity of Narrow-Bitwidth Functional Units needs to be balanced with respect to the overhead of bit-level aggregation as well as the width of vectorization.

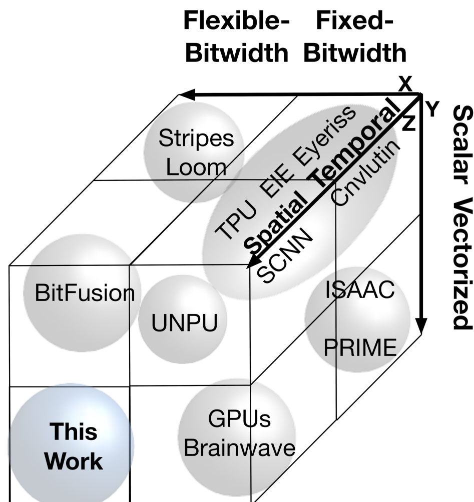


Figure 3.1. The landscape of DNN accelerators and how this work fits in the picture.

Additionally, the proposed approach creates a new opportunity for exploring bit-flexibility in the context of vectorized execution. As Figure 4.1 illustrates, the landscape of neural accelerators can be depicted in the three dimensions of Type of Functional Units (Scalar [55, 106, 29, 128, 184] v.s. Vectorized [215, 59, 90]), runtime Bit Flexibility (Fixed Bitwidth v.s. Flexible Bitwidth), and Composability (Temporal [129, 217, 152] vs. Spatial [222]). This work aims to fill the vacancy for Vectorized Bit-Flexible Spatial Composability. In fact, the interleaving of the bit-level parallelism with DLP creates the new opportunity to explore bit-level composability where a group of bit-parallel vector units *dynamically* form collaborative groups to carry out SIMD operations. Recent inspiring research investigates bit-level composability for a scalar operation (a single MAC unit) both in spatial and temporal mode. However, the space where composability (bit-level parallelism) meets vectorization (data-level parallelism) has not been thoroughly explored. Since the building block is a narrow bitwidth vector engine, another opportunity arises to support execution well below eight bits. That is leveraging the algorithmic insight that heterogeneous assignment of bitwidths below eight to DNN layers can reduce their computational complexity while preserving accuracy [117, 176, 61, 210, 82]. Fixed bitwidth designs cannot tap into this level of efficiency where each unit is only processing the fewest number of bits that can preserve the accuracy. BitBlade [207] proposes a similar design to compose

low-bitwidth vectorized operations to provide lower-cost bit-flexibility. In this work, we perform a holistic design analysis to well evaluate the potential of bit-parallel vector composability and extend its applicability to even efficient fixed-bit execution.

The evaluation of the proposed concepts are carried out with and without availability of algorithmic bitwidth heterogeneity. Experimentation with six real-world DNNs shows that bit-parallel vector composibility provides 40% speedup and energy reduction compared to design with the same architecture (systolic) without support for the proposed composability. When bit flexibility is applicable because of the heterogenous bitwidths in the DNN layers, our design provides 50% speedup and 10% energy reduction compared to BitFusion [222], the state-of-the-art architecture that supports scalar bit-level composability in systolic designs.

When a high bandwidth off-chip memory is utilized, the baseline design only enjoys 10% speedup and 30% energy reduction, respectively. However, bit-parallel vector-composability better utilizes the boosted bandwidth and provides 2.1$\times$ speedup and 2.3$\times$ energy reduction. With algorithmic bitwidth heterogeneity our design style provides 2.4$\times$ speedup and 20% energy reduction, compared to BitFusion, while it also utilizes the same high bandwidth memory Finally, we compare different permutations of our design style with respect to homogenous/heterogenous bitwidth and off-chip memory bandwidth to the Nvidia's RTX 2080 TI GPU which also supports INT-4 execution. Benefits range between 28.0$\times$and 33.7$\times$higher Performance-per-Watt for four possible design points.

3.2 Bit-Parallel Vector Composability

This work builds upon the fundamental property of *vector dot-product operation* – the most common operation in DNNs – that vector dot-product with wide-bitwidth data types can be decomposed and reformulated as a summation of several dot-products with narrow-bitwidth data types. The element-wise multiplication in vector dot-product can be performed independently, exposing *data-level parallelism*. This work explores another degree of parallelism

– *bit-level parallelism* (BLP) – wherein individual multiplications can be broken down at *bit-level* and written as a summation of narrower bitwidth multiplications. Leveraging this insight, this work studies the interleaving of both data-level parallelism in vector dot-product with bit-level parallelism in individual multiplications to introduce the notion of *bit-parallel vector composability*. This design style can also be exploited to support *runtime-flexible* bitwidths on the underlying hardware. The rest of this section details the mathematical formulation for bit-parallel vector composability.

Fixed-Bitwidth bit-parallel vector composability. Digital values can be expressed as the summation of individual bits multiplied by powers of two. Hence, a vector dot-product operation between two vectors, $\vec{X}$, $\vec{W}$ can be expressed as follows:

$$\vec{X} \bullet \vec{W} = \sum_i (x_i \times w_i) = \sum_i ((\sum_{j=0}^{bw_x-1} 2^j \times x_i[j]) \times (\sum_{k=0}^{bw_w-1} 2^k \times w_i[k])) \tag{3.1}$$

Variables bw_x and bw_w represent the bitwidths of the elements in $\vec{X}$ and $\vec{W}$, respectively. Expanding the bitwise multiplications between the elements of the two vectors yields:

$$\vec{X} \bullet \vec{W} = \sum_i \underline{(\sum_{j=0}^{bw_x-1} \sum_{k=0}^{bw_w-1} 2^{j+k} \times x_i[j] \times w_i[k])} \tag{3.2}$$

Conventional architectures rely on compute units that operate on all bits of individual operands, and require complex left-shift operations followed by wide-bitwidth additions as shown with an underline in Equation 3.2. By leveraging the associativity property of the multiplication and addition, we can cluster the bit-wise operations that share the same significance position together and factor out the power of two multiplications. In other words, this clustering can be realized by swapping the order of $\sum_i$ and $\sum_j \sum_k$ operators.

$$\vec{X} \bullet \vec{W} = \sum_{j=0}^{bw_x-1} \sum_{k=0}^{bw_w-1} 2^{j+k} \times \underline{(\sum_i x_i[j] \times w_i[k])} \tag{3.3}$$

Leveraging this insight enables the use of significantly less complex, narrow-bitwidth, compute units (1-bit in the equation), exploiting bit-level parallelism, amortizing the cost of left-shift and wide-bitwidth addition. Breaking down the dot-product is not limited to single bit and elements of the vectors can be *bit-sliced* with different sizes. As such, Equation 3.3 can be further generalized as:

$$= \sum_{j=0}^{\frac{bw_X}{\alpha}-1} \sum_{k=0}^{\frac{bw_W}{\beta}-1} 2^{\alpha j+\beta k} \times \underline{\left(\sum_i x_i[\alpha j : (\alpha+1)j] \times w_i[\beta k : (\beta+1)k]\right)} \tag{3.4}$$

Here, α and β are the bit-slices for operands x_i and w_i for the narrow-bitwidth compute units, respectively. Figure 3.2-(a) graphically illustrates bit-parallel vector composability using an example of a vector dot-product operation ($\vec{X} \bullet \vec{W} = \sum_i x_i \times w_i$) between two vectors of $\vec{X}$ and $\vec{W}$, each of which constitutes two 4-bit elements. As it is shown with different shades, each element can be bit-sliced and broken down into two 2-bit slices. With this bit-slicing scheme, the original element (x_i or w_i) can be written as $2^2 \times bsl_{MSB} + 2^0 \times bsl_{LSB}$, where bsls are the bit-slices and they are multiplied by powers of two based on their significance position. With the aforementioned bit-slicing scheme, performing a product operation between an element from $\vec{X}$ and another from $\vec{W}$ requires four multiplications between the slices, each of which is also multiplied by the corresponding significance position factor. However, because of the associativity property of the multiply-add, we cluster the bit-sliced multiplications that share the same significance position and apply the power of two multiplicands by shifting the accumulated result of the bit-sliced multiplications.

Flexible-Bitwidth vector composability. The bit-parallel vector composable design style can enable *flexible-bitwidth support at runtime*. Figure 3.2-(b) shows an example of flexible-bitwidth vector dot-product operation considering *the same number of compute resources (2-bit multipliers, adders, and shifters)* as Figure 3.2. In Figure 3.2-(b), a vector dot-product operation between a vector of inputs ($\vec{X}$) that has four 4-bit elements and a vector of weights ($\vec{W}$) that has

four 2-bit elements is illustrated. Using the same bit-slicing scheme, the original vector $\vec{X}$ is broken down to two sub-vectors that are required to go under dot-product with $\vec{W}$ and then get shifted and aggregated. However, exploiting 2-bit datatypes for weights compared to 4-bit in the example given in Figure 3.2-(a), provides 2× boost in compute performance. Bit-parallel vector composability enables maximum utilization of the compute resources, resulting in computing a vector dot-product operation with 2× more elements using the same amount of resources.

The next section discusses the acceleration using this design style.

3.3 Architecture Design for Bit-Parallel Vector Composability

To enable hardware realization for the insight of bit-parallel vector composability, the main building block of our design becomes a Composable Vector Unit (CVU), which performs the vector dot-product operation by splitting it into multiple *narrow-bitwidth* dot-products. As such, the CVU consists of several Narrow-Bitwidth Vector Engines (NBVE) that calculate

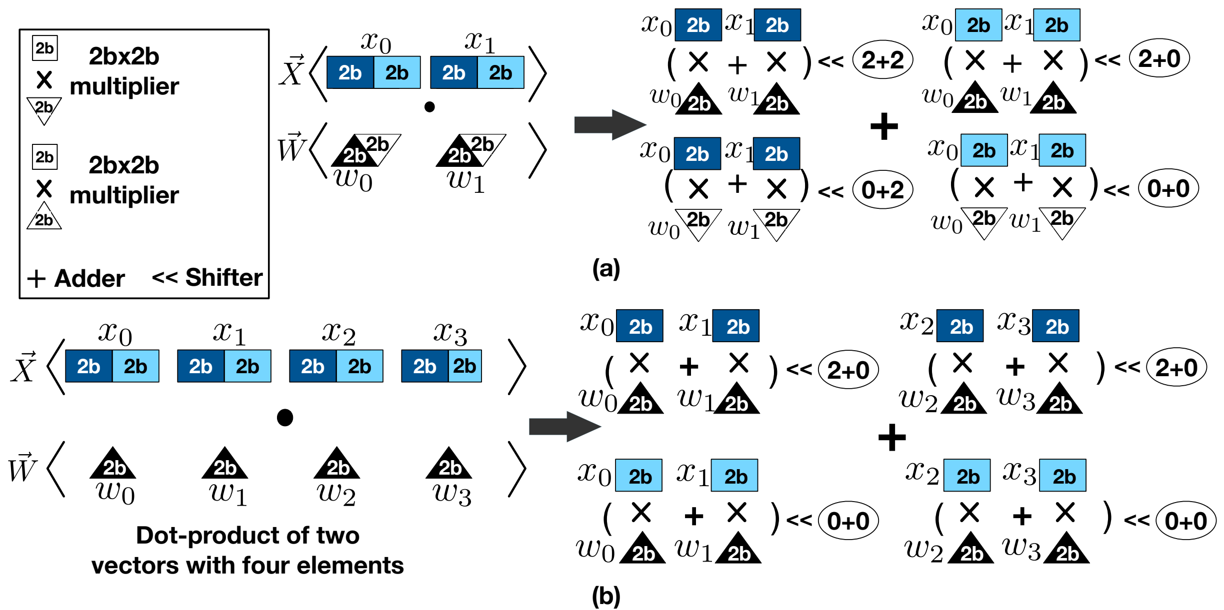


Figure 3.2. (a) Fixed-bitwidth bit-parallel vector composability with 2-bit slicing and (b) Bit-Flexible vector composability for 4-b inputs and 2-b weights; 2x improvement in performance compared to fixed 4-bit dot-product.

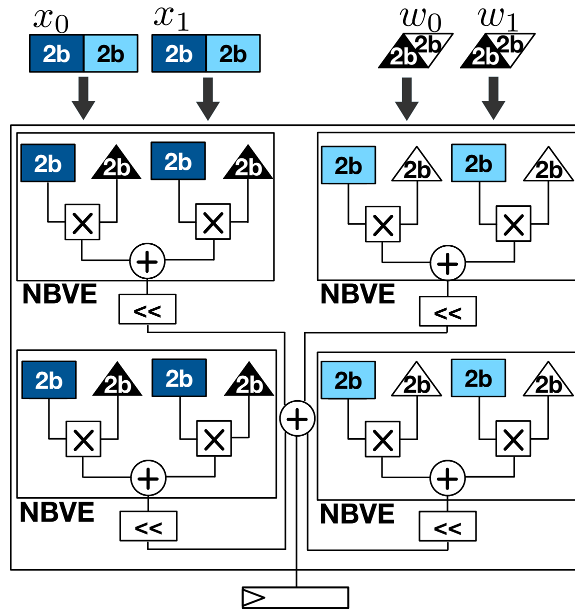
the dot-product of bit-sliced sub-vectors from the original vectors. The CVU then combines the results from NBVEs according to the bitwidth of each DNN layer. Below, we discuss the micro-architecture.

3.3.1 Composable Vector Unit (CVU)

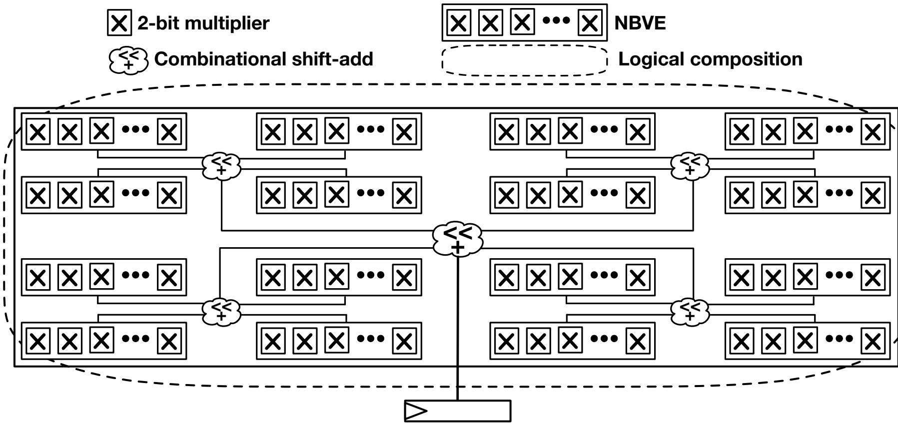
Figure 3.3-(a) illustrates an instance of the hardware realization of CVU for the vector dot-product example given in Figure 3.2-(a). In this example, CVU encapsulates 4 Narrow-Bitwidth Vector Engines (NBVE). The number of NBVEs inside a CVU is based on the size of bit-slicing and maximum bitwidth of datatypes (inputs and weights) that have been selected as two and four in this example, respectively. A spatial array of narrow-bitwidth multipliers, connected through an adder-tree, constitute each NBVE. Each narrow-bitwidth multiplier performs a 2-bit×2-bit multiplication with a 2-bit-slice of inputs and weights, the result of which then goes to the adder-tree to get aggregated with the outputs of other multipliers inside an NBVE. As such, the NBVE generates a single scalar that is the result of the dot-product operation between two bit-sliced sub-vectors. The CVU then shifts the outputs of NBVEs based on the significance position of their bit-sliced operands and aggregates the shifted values across all the NBVEs to generate the final result of the vector dot-product operation. In our design, we consider 8-bit as the maximum bitwidth of inputs and weights, commensurate with prior work [128], and 2-bit slicing for the CVU. As such, CVU encapsulates 16 NBVEs that work in parallel. Below, we discuss how the CVU operates when homogenous and heterogenous bitwidths are exploited.

Homogeneous 8-bit mode of operation. Figure 3.3(b) illustrates the conceptual view of CVU when 8-bit datatypes are homogeneously exploited for DNN layers. Each NBVE performs a bit-parallel dot-product on 2-bit-sliced sub-vectors of the original 8-bit×8-bit dot-product between vectors of length L, generating a scalar. The NBVEs are equipped with shifters to shift their scalar outputs. Finally, to generate the final scalar of the 8-bit dot-product, all the NBVEs in a CVU, globally cooperate together and CVU aggregates their outputs.

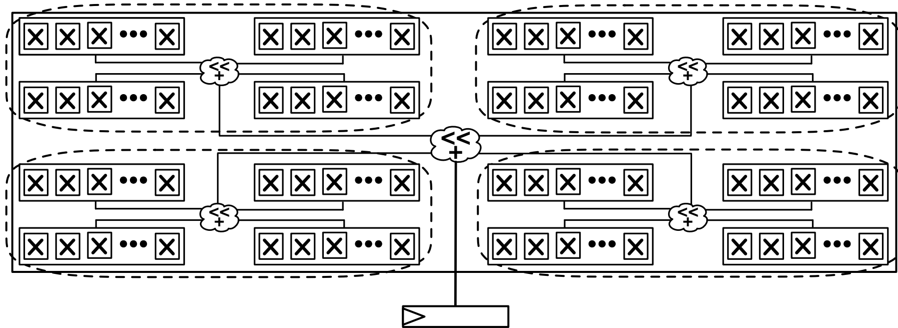
Heterogeneous quantized bitwidth mode of operation. When the DNN uses heterogeneous



(a) Hardware realization of CVU for the example in Figure 2-(a)



(b) Homogenous 8-bit mode, conceptual view; All the NBVEs cooperate together and perform an 8-bit×8-bit dot product with length of L



(c) Heterogenous quantized bitwidth mode, conceptual view; four clusters of NBVEs work in parallel and then cooperate together to perform an 8-bit×2-bit dot-product with length of 4×L

Figure 3.3. Composable Vector Unit.

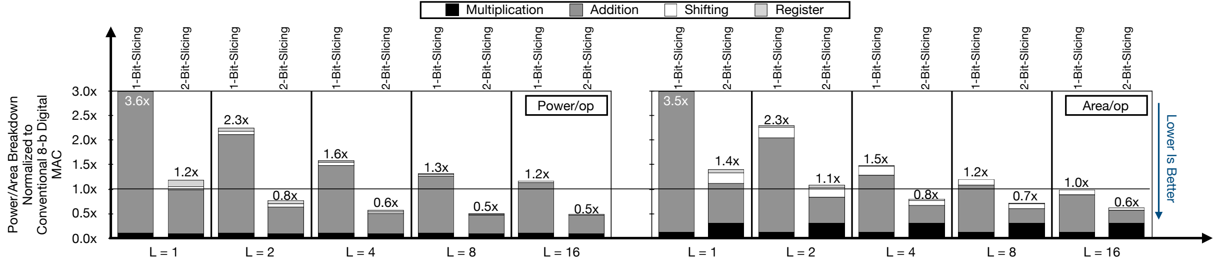


Figure 3.4. Design space exploration for size of bit-slicing and vector lengths for vector composability.

bitwidths (less than 8-bit datatypes) across its layers, the CVU can be dynamically reconfigured to match the bitwidths of the DNN layers at runtime. This includes both the reconfigurations of the shifters and the NBVEs composition scheme. For instance, Figure 3.3 (c) illustrates a case when 8-bit inputs and 2-bit weights are used, the 16 NBVEs will be clustered as four groups, each of which encapsulates four NBVEs. In this mode, the CVU composes the NBVEs in two levels. At the first level, all the four NBVEs in each cluster are privately composed together by applying the shift-add logic to complete a dot-product operation with a length of L. At the second level, the CVU globally aggregates the outputs of all four clusters and produces the scalar result of dot-product operation between vectors of length $4 \times L$. As another example, if 2-bit datatypes are used for both inputs and weights, each NBVE performs an independent dot-product, providing $16\times$ higher performance compared to the homogenous 8-bit mode. BiBlade[207], also proposes a very similar design, where the shift-add logic is shared across a group of low-bitwidth multipliers, to amortize the aggregation cost. In this work, to evaluate the benefits of the bit-parallel vector composability and the design tradeoffs corresponding to various architectural parameters, we perform a design space exploration for different number of multipliers in an NBVE (L) and choice of bit-slicing and analyze the sensitivity of the CVU's power/area to these parameters, as follows.

3.3.2 Design Space Exploration and Tradeoffs

Figure 3.4 shows the design space exploration for 1-bit and 2-bit slicing, in addition to different lengths of vectors for NBVE from $L = 1$ to $L = 16$. All the design points in this

analysis are synthesized in 500 Mhz and 45 nm node. Y-Axis shows the power and area per one 8-bit $\times$ 8-bit MAC operation performed by CVU normalized to the power/area of a conventional digital 8-bit MAC unit. We sweep the L parameter for 1-bit slicing and 2-bit slicing in the X-axis. Figure 3.4 also shows the breakdown of power and area across four different hardware logics; multiplication, addition, shifting, and registering. Inspecting this analysis leads us to following key observations:

(1) Adder-tree consumes the most power/area and might bottleneck the efficiency. As we observe in both 1-bit and 2-bit slicing, across all the hardware components adder-tree ranks first in power/area consumption. Bit-parallel vector composability imposes two levels of add-tree logic: (a) An adder-tree private to each NBVE that sums the results of narrow-bitwidth multiplications. (b) A global adder-tree that aggregates the outputs of NBVEs to generate the final scalar result. Hence, to gain power/area efficiency, the cost of add-tree requires to be minimized.

(2) Integrating more narrow bitwidth multipliers within NBVE*s (exploiting DLP within BLP) minimizes the cost of add-tree logic.* Encapsulating a larger number of narrow-bitwidth multipliers in an NBVE leads to amortizing the cost of add-tree logic across a wider array of multipliers and yields power/area efficiency. As it is shown in Figure 3.4, increasing L from 1 to 16 improves the power/area by $\approx 3\times$ in 1-bit slicing and $\approx 2.5\times$ in 2-bit slicing. However, this improvement in power/area gradually saturates. As such, increasing L beyond 16 does not provide further significant benefits.

(3) The 2-bit slicing strikes a better balance between the complexity of the narrow-bitwidth multipliers and the cost of aggregation and operand delivery in CVU*s.* 1-bit slicing requires 1-bit multipliers (merely AND gates), that also generates 1-bit values as the inputs of the adder-trees in NBVEs. However, slicing 8-bit operands to 1-bit require $8 \times 8 = 64$ NBVEs for a CVU, imposing a costly 64-input global adder-tree to CVUs. Consequently, as it is shown in Figure 3.4, 1-bit slicing does not provide any benefits compared to the conventional design. On the other hand, although 2-bit slicing results in generating 4-bit values by the multipliers as inputs to adder-trees in NBVEs, it quadratically decreases the total number of NBVEs in a CVU from

64 to 16. This quadratic decrease trumps using wider-bitwidth values as the inputs of adder-trees in NBVEs, significantly lowering add-tree cost. In conclusion, the optimal design choice for a digital CVU comes with 2-bit slicing and length of $L = 16$. Compared to a conventional digital design, this design point provides 2.0× and 1.7× improvement in power and area respectively, for an 8-bit × 8-bit MAC operation. Note that 4-bit slicing provides lower power/area for CVU design, however, it leads to underutilization of compute resources when DNNs with less than 4-bits are being processed. As such, 2-bit slicing strikes a better balance between the efficiency of CVUs and their overall utilization.

(4) Bit-parallel vector composability amortizes the cost of flexibility across the elements of vectors. Prior bit-flexible works, both spatial (e.g., BitFusion [222]) and temporal (e.g., Stripes and Loom [129, 217]), enable supporting deep quantized DNNs with heterogenous bitwidths at the cost of extra area overheads. BitFusion [222] exploits spatial bit-parallel composability for a scalar. This design can be assumed as one possible configuration of bit-parallel vector composability with 2-bit slicing and $L = 1$. As shown in Figure 3.4, this design point imposes 40% area overhead as compared to conventional design, while our design provides 40% reduction in area. Also our proposed CVU provides 2.4× improvement in power as compared to Fusion Units in BitFusion. This result seems counter intuitive at the first glance as flexibility often comes with an overhead. In fact, the cost of bit-level flexibility stems from aggregation logic that puts the results back together. Our proposed bit-parallel vector-level composability amortizes this cost across the elements of the vector. Moreover, since it reduces the complexity of the cooperating narrower bitwidth units, it leads to even further reduction.

3.3.3 Overall Architecture

Conceptually, bit-parallel vector composability is orthogonal to the architectural organization of the CVUs. We explore this design style using a 2D systolic array architecture, which is efficient for matrix-multiplications and convolutions, as explored by prior works [128]. In this architecture, called BPVEC, each CVU reads a vector of weights from its private scratchpad,

Table 3.1.
Evaluated DNN models.

DNN Models	Type	Model Size (INT8)	Multiply-Adds (GOps)	Heterogenous Bitwidths
AlexNet	CNN	56.1 MB	2,678	First and last layer 8-bit, the rest 4-bit
Inception-v1	CNN	8.6 MB	1,860	First and last layer 8-bit, the rest 4-bit
ResNet-18	CNN	11.1 MB	4,269	First and last layer 8-bit, the rest 4-bit
ResNet-50	CNN	24.4 MB	8,030	All layers with 4-bit
RNN	RNN	16.0 MB	17	All layers with 4-bit
LSTM	RNN	12.3 MB	13	All layers with 4-bit

while a vector of inputs is shared across columns of CVUs in each row of the 2D array. The scalar outputs of CVUs aggregate across columns of the array in a systolic fashion and accumulate using 64-bit registers.

3.4 Evaluation

3.4.1 Methodology

Workloads. Table 3.1 details the specification of the evaluated models. We evaluate our proposal using these neural models in two cases of homogenous and heterogenous bitwidths. For the former one we use 8-bit datatypes for all the activations and weights and for the later we use the bitwidths reported in the results of the literature [176, 117, 61] that maintain the full-precision accuracy of the models.

ASIC baselines. For the experiments with homogeneous fixed bitwidths, we use a TPU-like accelerator with a systolic architecture. For the case of heterogeneous bitwidths, we use BitFusion [222], a state-of-the-art spatial bit-flexible DNN accelerator, as the comparison point. In all setups, we use 250 mW core power budget for all the baselines and the proposed accelerator in 45 nm technology node and with 500 Mhz frequency. Table 3.2 details the specifications of the evaluated platforms. We modify the open-source simulation infrastructure in [222] to obtain end-to-end performance and energy metrics for the TPU-like baseline accelerator, baseline

Table 3.2.
Evaluated hardware platforms.

	ASIC Platforms			GPU Platform	
Chip	TPU-Like Baseline	BitFusion	BPVeC	Chip	RTX 2080 TI
# of MACs	512	448	1024	# of Tensor Cores	544
Architecture	Systolic	Systolic	Systolic	Architecture	Turing
On-chip memory	112 KB	112 KB	112 KB	Memory	11 GB (GDDR6)
Frequency	500 Mhz	500 Mhz	500 Mhz	Frequency	1545 Mhz
Technology Node	45 nm	45 nm	45 nm	Technology Node	12 nm

BitFusion accelerator, as well as the proposed BPVEC accelerator.

GPU baseline. We also compare BPVEC to the Nvidia's RTX 2080 TI GPU, equipped with tensor cores that are specialized for deep learning inference. Table 3.2 shows the architectural parameters of this GPU. For the sake of fairness, we use 8-bit execution for the case of homogenous bitwidths and 4-bit execution for heterogenous bitwidths using the Nvidia's TensorRT 5.1 compiled with CUDA 10.1 and cuDNN 7.5.

Hardware measurements. We implement the proposed accelerator using Verilog RTL. We use Synopsis Design Compiler (L-2016.03-SP5) for synthesis and measuring energy/area. All the synthesis for the design space exploration presented in Figure 3.4 are performed in 45 nm technology node and 500 Mhz frequency and all the design points meet the frequency criteria. The on-chip scratchpads for ASIC designs are modeled with CACTI-P.

Off-chip memory. We evaluate our design style with both a moderate and high bandwidth off-chip memory system to assess its sensitivity to the off-chip bandwidth. For moderate bandwidth, we use DDR4 with 16 GB/sec bandwidth and 15 pJ/bit energy for data accesses. We model the high bandwidth memory based on HBM2 with 256 GB/sec bandwidth and 1.2 pJ/bit energy for data accesses [183].

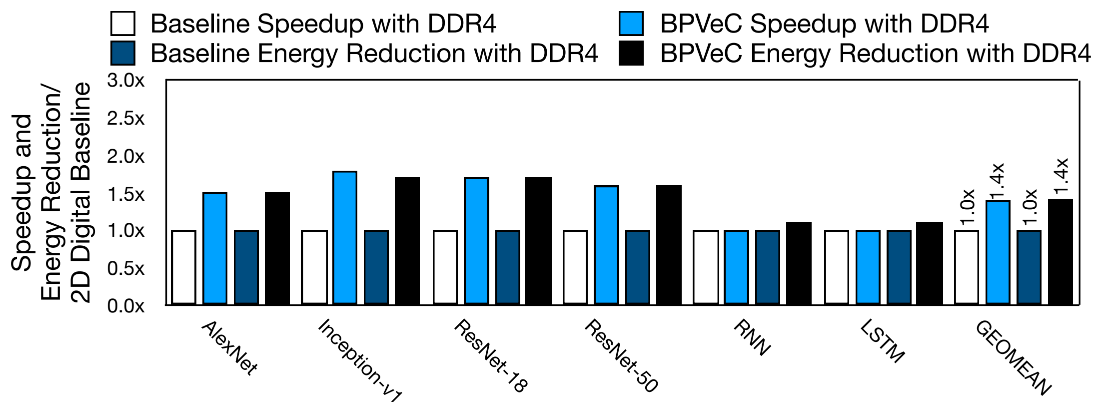


Figure 3.5. Comparison to baseline; DDR4 memory and without bitwidth heterogeneity.

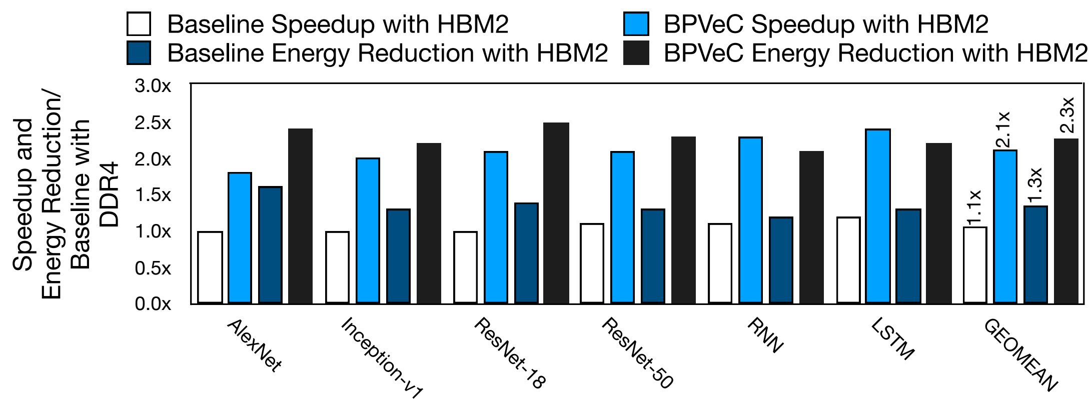


Figure 3.6. Comparison to baseline; HBM2 memory and without bitwidth heterogeneity.

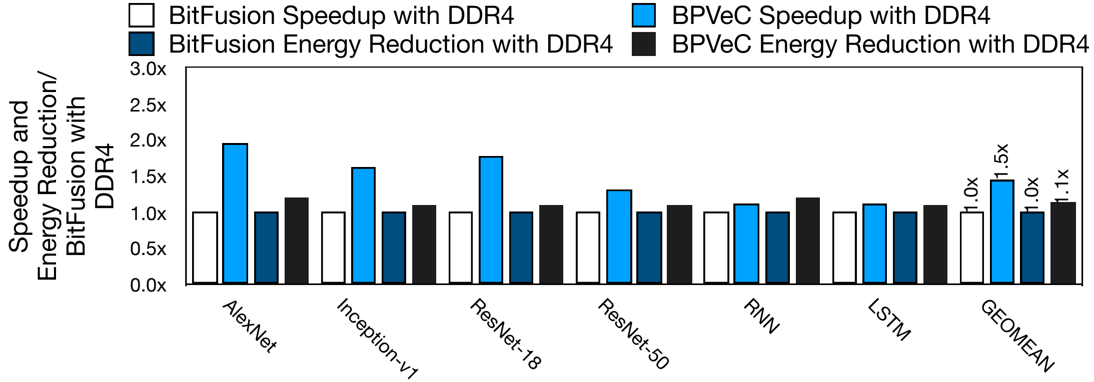


Figure 3.7. Comparison to BitFusion; DDR4 memory and with bitwidth heterogeneity.

3.4.2 Experimental Results

Without bitwidth heterogeneity. Figure 3.5 evaluates the performance and energy of BPVEC across a range of DNNs with homogenous btiwidths (8-bit). The baseline uses the same systolic-array architecture as the BPVEC accelerator, but with conventional MAC units. Both designs use a DRR4 memory system. Bit-parallel vector composability enables our accelerator to integrate $\approx 2.0\times$ more compute resources compared to the baseline design under the same core power budget. On average, BPVEC provides 40% speedup and energy reduction. Across the evaluated workloads, CNN models enjoy more benefits compared to RNN ones. Unlike the CNNs that have significant data-reuse, the vector-matrix multiplications in RNNs have limited data-reuse and require extensive off-chip data accesses. As such, the limited bandwidth of the DDR4 memory leads to starvation of the copious on-chip compute resources in BPVEC.

Figure 3.6 compares benefits from a high bandwidth memory (HBM2) for the baseline and BPVEC, normalized to the baseline with DDR4. While benefits baseline design see limited benefits, our BPVEC enjoys a $2.1\times$and $2.3\times$speedup and energy reduction, respectively. Results suggest that BPVEC is better able to exploit the increased bandwidth and reduced energy cost from HBM2 memory system. While all benchmarks see improved efficiency, benefits are highest for bandwidth-hungry RNN and LSTM.

With bitwidth heterogeneity. Figure 3.7 evaluates the performance and energy benefits for

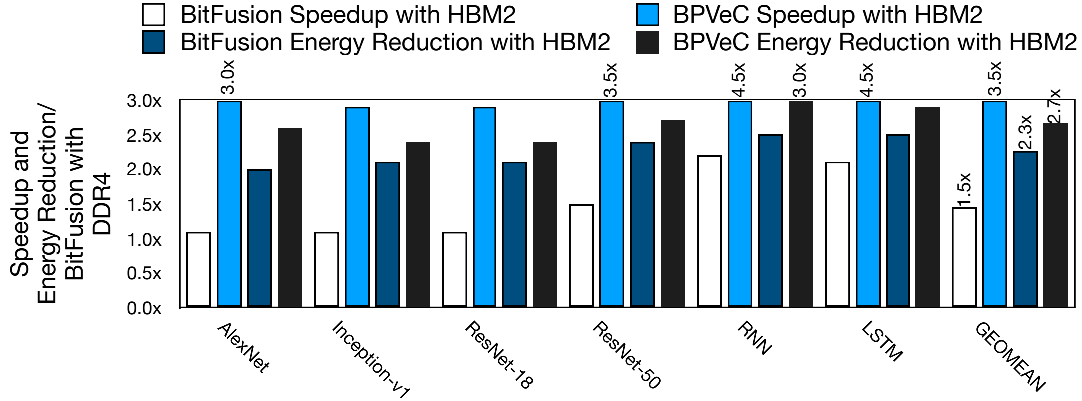
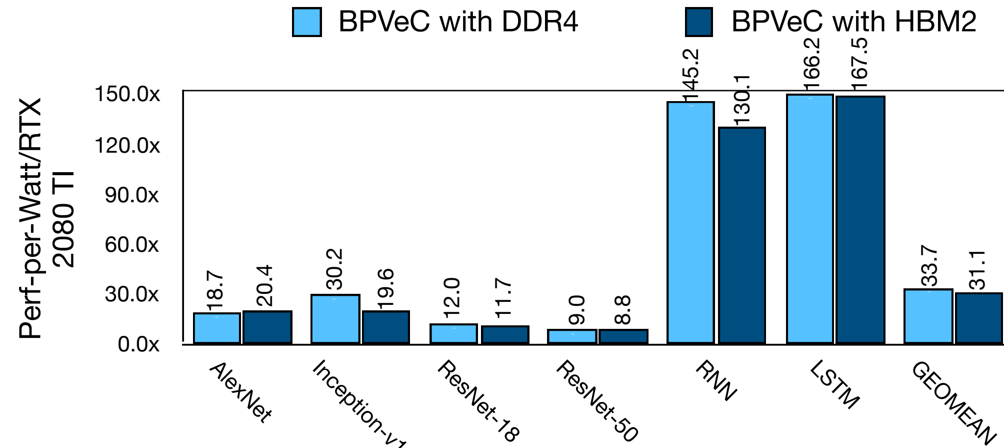


Figure 3.8. Comparison to BitFusion; HBM2 memory and with bitwidth heterogeneity.

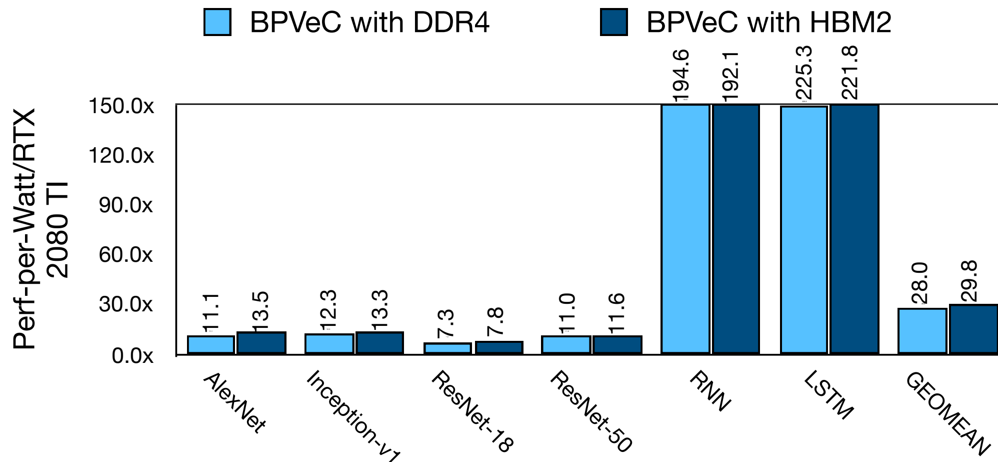
quantized DNNs with heterogenous bitwidths. The baseline in Figure 3.7 is BitFusion [222], a state-of-the-art accelerator that also supports flexible bitwidths, but at the scalar level. In this experiment, the baseline BitFusion and BPVEC use DDR4 memory. Bit-parallel vector composability enables our design to integrate $\approx 2.3\times$ more compute resources compared to BitFusion under the same core power budget. On average, BPVEC provides 50% speedup and 10% energy reduction over BitFusion. Across the evaluated workloads, CNN models enjoy more benefits compared to bandwidth-hungry RNN and LSTM.

Figure 3.8 studies the interplay of high off-chip bandwidth with flexible-bitwidth acceleration. The speedup and energy reduction numbers are normalized to BitFusion with moderate bandwidth DDR4. BPVEC provides $2.5\times$speedup and 20% energy reduction over BitFusion with HBM2 memory ($3.5\times$ speedup and $2.7\times$ energy reduction over the baseline 2D BitFusion). RNN and LSTM, see the highest performance benefits ($4.5\times$), since these benchmarks can take advantage of both the increased compute units in BPVEC design, as well as the increased bandwidth form HBM2.

GPU comparison. Figure 4.20 compares the Performance-per-Watt of BPVEC design with DDR4 and HBM2 memory and with homogenous (Figure 4.20 (a)) and heterogenous (Figure 4.20) bitwidths for DNN layers, respectively as compared to the Nvidia's RTX 2080 TI GPU. With homogenous bitwidths (Figure 4.20 (a)), BPVEC achieves $33.7\times$ and $31.1\times$ improvements



(a) Perf-per-Watt Comparison with RTX 2080 TI with homogenous bitwidths



(b) Perf-per-Watt Comparison with RTX 2080 TI with heterogenous bitwidths

Figure 3.9. Performance-Per-Watt comparison to RTX 2080 TI GPU.

on average with DDR4 and HBM2 memory, respectively. Across the evaluated workloads, the RNN models see the most benefits. These models require a large amount of vector-matrix multiplications, which particularly suitable for the proposed bit-parallel vector composability design style. In the case of heterogenous bitwidths, the benefits go to 28.0× and 29.8× with DDR4 and HBM2, respectively. The trends look similar to the homogenous 8-bit mode, since both the design points and the GPU baseline exploit the deep quantized mode of operations.

3.5 Related Work

A large body of inspiring work has explored hardware acceleration for DNNs by exploiting their algorithmic properties such as data-level parallelism, tolerance for reduced precision and sparsification, and redundancy in computations. To realize the hardware accelerators, prior efforts have built upon isolated compute units that operate on all the bits of individual operands, and have used multiple compute units operating together to extract data-level parallelism. This work introduces a different design style that explores the interleaving of bit-level operations across compute units in the context of *vectors* and combine the benefits from bit-level parallelism and data-level parallelism, both of which are abundant in DNNs. Below, we discuss the most related works.

Design without support for bit-level composability. Prior works in TPU [128] and Eyeriss [55] design hardware accelerators to extract data-level parallelism in DNNs. SCNN [184], EIE [106], and Cnvlutin [29] use both zero-skipping and data-level parallelism for efficient DNN execution. Brainwave [90] also uses SIMD vectorized execution to extract data-level parallelism on FPGAs, however with fixed-bitwidth execution whose bitwidth is decided before synthesizing the FPGA on the design. ISAAC [215] and PRIME [59] build upon ResistiveRam(ReRam) technology to provide high energy efficiency. Further, ISAAC and PRIME operate on vectors of data in the analog (current) domain to mitigate the high cost of ADC. In contrast, we focus on interleaving of the bit-level and data-level parallelism in vector units, in addition to hardware support for bitwidth heterogeneity.

Design with support for bit-level flexibility through bit-serial computation. Stripes [129], Loom [217], UNPU [152] exploit the tolerance to reduced bitwidth in DNNs to yield performance benefits by exploring bit-*bit serial compute units*. The data-level parallelism compensates for bit-serial individual operations. Our design in contrast interleaves bit-level parallelism with data-level parallelism.

Designs with support for bit-level composability BitFusion [222] uses bit-level parallelism

with a spatial design. We provide a head-to-head comparison with BitFusion in Section 3.4.2. Laconic [218] combines spatial bit-level composability with temporal execution to support for bit-sparsity and reduce the ineffectual computations. These inspiring efforts do not focus on bit-parallel vector composability that breaks the calculations across spatial composable units that cooperate at the level of vectors. BitBlade [207] proposes to share the shift-add logic in Fusion Units of BitFusion across a vector of elements to amortize its cost for bit-flexible execution, very similar to our bit-parallel vector composability. However, in this work, we provide a holistic design analysis considering the implications of various architectural parameters to well study the potential of bit-parallel vector composability and enable that also for optimized homogeneous fixed-bit execution.

3.6 Conclusion

Traditionally, neural accelerators have relied on extracting DLP using isolated and self-sufficient compute units that process all the bits of operands. This work introduced a different design style, *bit-parallel vector-composability*, that operates on operand bit-slices to interleave and combine the traditional data-level parallelism with bit-level parallelism. Across a range of deep models the results show that the proposed design style offers significant performance and efficiency compared to even bit-flexible accelerators.

3.7 Acknowledgement

Chapter 3 is a partial reprint of the material as it appears in: S. Ghodrati, H. Sharma, C. Young, N. Kim, and H. Esmaeilzadeh, “Bit-Parallel Vector Composability for Neural Acceleration.” in *Design Automation Conference (DAC)*, 2020. The dissertation author was the primary investigator and author of this paper.

Chapter 4

Balancing Specialization and Programmability for Efficient End-to-End Acceleration of Deep Neural Networks

4.1 Introduction

Deep Neural Networks (DNNs) have taken the IT industry and almost every computing research community by storm. Their compute intensity has heralded an era of neural accelerators or neural processing units [295, 106, 29, 184, 147, 27, 57, 191, 103, 261, 129, 222, 28, 220, 74, 219, 206, 215, 59, 31, 286, 158, 233, 97, 135, 80, 119, 92, 58, 55, 221, 167, 128, 112, 90, 94, 111, 216, 145, 79, 85, 178, 39, 229, 291, 185, 169, 293]. These designs started with mostly focusing on convolutions, then later on more broadly on GEneral Matrix Multiplication (GEMM) operations as they dominated the neural networks initially. Researchers have focused on optimizing the design for these GEMM operations from various aspects including but not limited to sparsification [295, 106, 29, 184, 147, 27, 57, 191, 103, 261], bit-level flexibility [129, 222, 28, 220, 74, 219, 206], use of resistive technologies [215, 59, 228, 31, 286], analog computations [158, 233, 97], in/near memory computation [135, 80, 119, 92], data flow optimizations [92, 58, 55, 221, 167, 128, 112, 90, 94, 111, 216, 145], to name a few. These inspiring innovations have been effective in optimizing the runtime and energy efficiency of GEMM-based operations. However, the scale is shifting as the time passes. As illustrated in

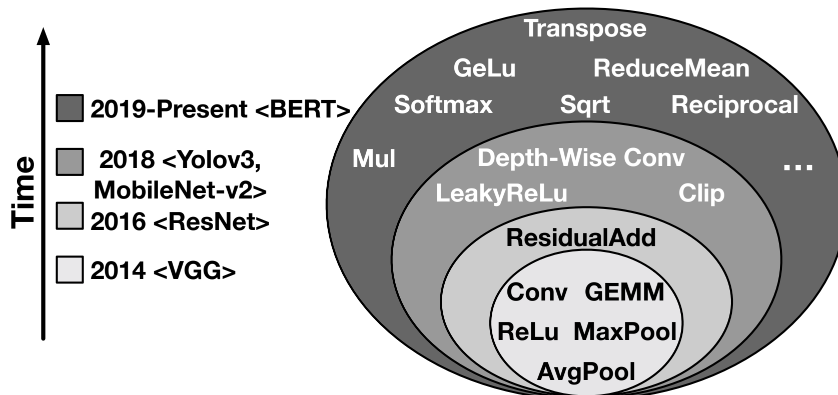


Figure 4.1. Union set of neural operators/layers in representative DNNs over the years.

Figure 4.1, non-GEMM operations have increased significantly in number, variety, and the structure of connectivity. For instance, in VGG-16 [224], as the first generation of DNNs, ∼70% of the total operators are non-GEMM that are from only three types. Whereas in Transformers (e.g., BERT [78]), as the current generation, ∼90% are non-GEMMs, mostly from ten kinds. This trend is expected to continue as DNNs enter more domains.

The non-GEMM operations are traditionally delegated to a few dedicated blocks (e.g., the ReLu/MaxPool units) [58, 106, 215, 59, 158, 221, 129, 184, 79, 147, 15, 80, 90, 222, 167, 112, 178, 27, 39, 229, 31, 216, 145, 97]. However, this approach is not sustainable as the variety of the non-GEMM operations and their structural connectivity to other layers increase. Clearly, there is a need for a rather significant degree of programmability. As such, alternative to or in addition to these blocks, an off-chip general-purpose processor [55, 92, 291, 29, 28, 185, 169, 111, 74, 57, 94, 219, 293, 191, 276, 232] or an on-chip one [182, 128, 127, 247, 95, 239] is designated to handle non-GEMM operations. Through our evaluations, we observe that runtime effects of the non-GEMM operations grow in dominance and they are no longer a rather small and limited minority. Their runtime effects are amplified as the GEMM unit has been polished and optimized over the past decade. Due to these optimizations, Amdahl's bottleneck is shifting towards these non-GEMM operations. Moreover, the non-GEMM counterpart needs to keep up with this optimized GEMM unit to sustain both of their utilization levels.

To address these emerging challenges, this work proposes a third alternative: a specialized,

yet programmable processor, which acts as a companion to the GEMM unit. This specialized processor, named the Tandem Processor, not only handles the execution of the non-GEMM layers, but also orchestrates the end-to-end DNN execution and operand delivery between units. To strike a balance between specialization and programmability, on the one hand, we specialize its ISA and memory semantics and alleviate the register file and its associated load/store operations. These specializations are derived from the common patterns of accesses in non-GEMM layers that loop to rearrange and process data elements for the next layer. On the other hand, the calculations of the non-GEMM layers are only supported through primitive arithmetic/logic vector operations. Therefore programmability is offered at the mathematical level.

Contributions:

(1) This research explores the uncharted and rather ignored non-GEMM layers and their challenging structural and computational effects on the end-to-end DNN acceleration.

(2) We leverage the unique characteristics of non-GEMM layers and propose a new instruction execution semantic and architecture (the Tandem Processor) that does not adhere to the conventional Register-File-centric designs. This design choice enables an exclusively specialized data access semantic that is unique to the Tandem Processor.

(3) Furthermore, we leverage the common data manipulation patterns in non-GEMM DNN layers and offer a pipeline front-end that leverages microarchitectural mechanisms to keep track of strided iterators. This design innovation minimizes the overhead of loop execution, address calculations, and memory accesses.

(4) Finally, we provide the Verilog code and the associated compiler as artifacts that will be open-sourced and we also present the chip floorplan and post-layout analysis.

We evaluate the Tandem Processor with respect to ***end-to-end***[1] execution of six diverse DNNs, when Tandem Processor or the alternative design points augment the same GEMM unit. The results show that a balanced design offers significant advantages (2.9× speedup and 20.1×

[1]The end-to-end implies the execution of both GEMM and non-GEMM layers.

energy reduction) over the common practices of using dedicated blocks that may also require help from the host processor. In an iso-resource setting, we compare the Tandem Processor to a recent inspiring academic project [95] that uses an on-chip RISC-V processor in addition to the dedicated blocks. Utilizing Tandem Processor outperforms the use of on-chip multi-core RISC-V processor by 5.3×. Using Tandem Processor even surpasses an impractical design that integrates a high-end Intel multi-core CPU with AVX support within the neural accelerator (2.3× speedup on average and 37.5× energy reduction). Finally, comparison with NVIDIA's Jetson Xavier NX GPU that leverages NVDLA accelerator [15] shows 4.1× improvements in performance-per-Watt with ∼12× less resources.

4.2 Diving Deeper into non-GEMM Operations

Table 4.1. Non-GEMM operators and their representative DNNs.

Non-GEMM Operator Classes	Operator Examples	Representative DNNs
Element-wise mathematical operators	Add, Sub, Mul, Exp, Sqrt, Floor, Ceil, Greater, Equal, Less, Pow, Reciprocal	ResNet [110], Yolov3 [199], MobileNetv2 [212], EfficientNet [244], BERT [78]
Element-wise activation function	Relu, LeakyRelu, Clip, Tanh, Sigmoid, GeLU	VGG-16 [224], ResNet [110], Yolov3 [199], MobileNetv2 [212], EfficientNet [244], BERT [78]
Reduction-based operators	Depth-wise Conv, MaxPool, GlobalAveragePool, ReduceMean, Softmax	VGG-16 [224], ResNet [110], MobileNetv2 [212], EfficientNet [244], BERT [78]
Data layout transformation	Transpose, Reshape, Concat	Yolov3 [199], BERT [78]
Type conversion	Cast, BitShift	Any Inference

This section first characterizes the emerging non-GEMM operations in DNNs. It then discusses requirements for supporting these operations in DNN accelerators, driven by their characteristics. Finally, it goes over the prior approaches in supporting these operations and qualitatively analyzes them with respect to the requirements.

4.2.1 Characteristics of Non-GEMM Operations in Modern DNNs

Non-GEMM operations are significantly diverse. Table 4.1 summarizes the non-GEMM operators used for inference across a set of diverse DNN models. We extract these operations from their corresponding ONNX implementations [174]. These layers can be categorized into five classes: (1) element-wise mathematical operations, (2) element-wise activation functions, (3) reduction-based operations, (4) data layout transformation operations, and (5) data type conversion operations. Non-GEMM operators fundamentally differ from GEMM ones. They

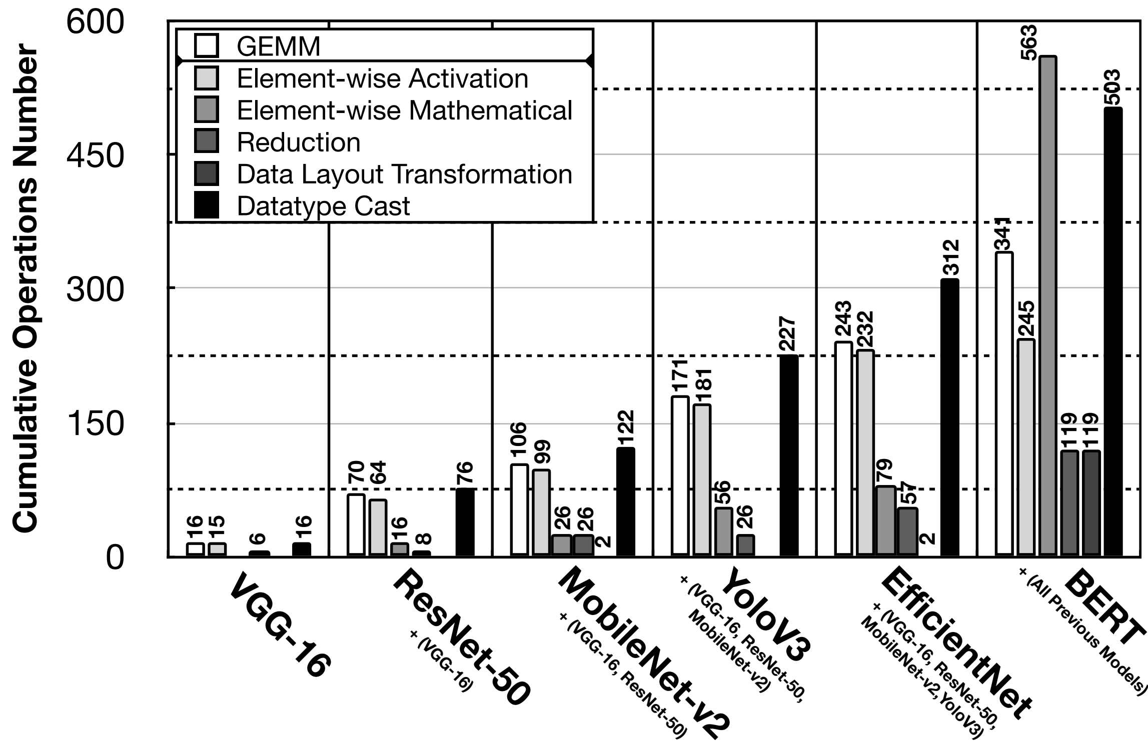


Figure 4.2. Cumulative number of GEMM and non-GEMM operations across benchmarks. Last bar covers the frequency of usage across all the models.

exhibit a wide diversity in terms of compute operations ranging from simple mathematical operations (e.g. Add, Mul, etc.) to complex ones (e.g. GeLU, Exp, etc.) as opposed to the commonly used multiply-accumulate in GEMM layers. Moreover, they require various patterns of mapping between input and output tensors, from one-to-one in element-wise operations to many-to-one in reduction-based ones.

Usage frequency of non-GEMM operators is continuously growing. Figure 4.2 shows the usage frequency of the GEMM and non-GEMM operators across the studied benchmarks. We extract this data from the ONNX graph representation of each model and categorize them with respect to the classification in Table 4.1. The y-axis shows the cumulative usage of these operators as additional models are taken into account[2]. The last group of bars show the total cumulative usage of operators across all benchmarks. As shown in Figure 4.2, as additional models are covered, the cumulative number of non-GEMM operations noticeably surges. Additionally,

[2]The models are added in chronological order based on their introduction time.

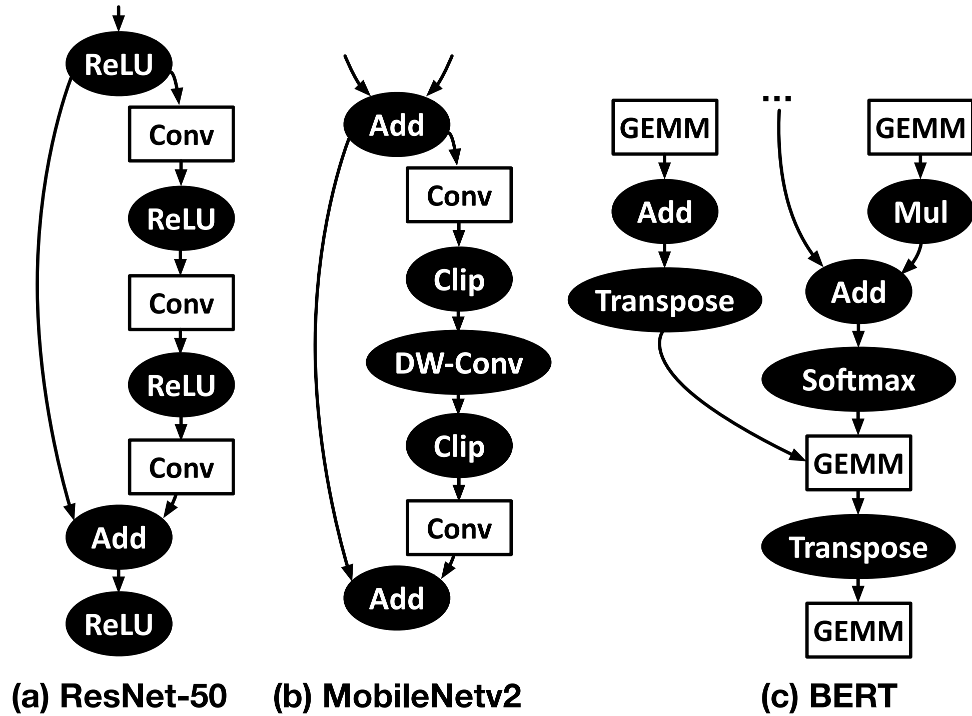


Figure 4.3. Repeated subgraphs of (a) ResNet-50 [110], (b) MobileNetv2 [212], and BERT [78]. The gray ovels illustrate the non-GEMM operations and white rectangles show the GEMM-based operations.

taking the entire benchmarks into account (last bar), merely 18% of total DNN operator nodes are GEMMs.

Non-GEMMs are interspersed amongst GEMMs. Figure 4.3 depicts the core subgraphs of three DNNs (ResNet-50, MobileNetv2, and BERT) which are frequently repeated in each model. As shown, the non-GEMM operators are interspersed amongst the GEMM ones (e.g. Conv) with various forms of connectivity. This structure demands iterative back-and-forth data exchange between GEMM and non-GEMM units that can happen through off-chip or on-chip memory. On top of this data exchange, tensor reformatting such as datatype casting and tensor layout transformations may be required. This is due to the differences between the execution semantics of the GEMM and non-GEMM units and their operands bitwidths.

The majority of non-GEMM operators are memory-bound. The majority of non-GEMM layers are element-wise operations (>80%). Moreover, the ones that are not element-wise exhibit

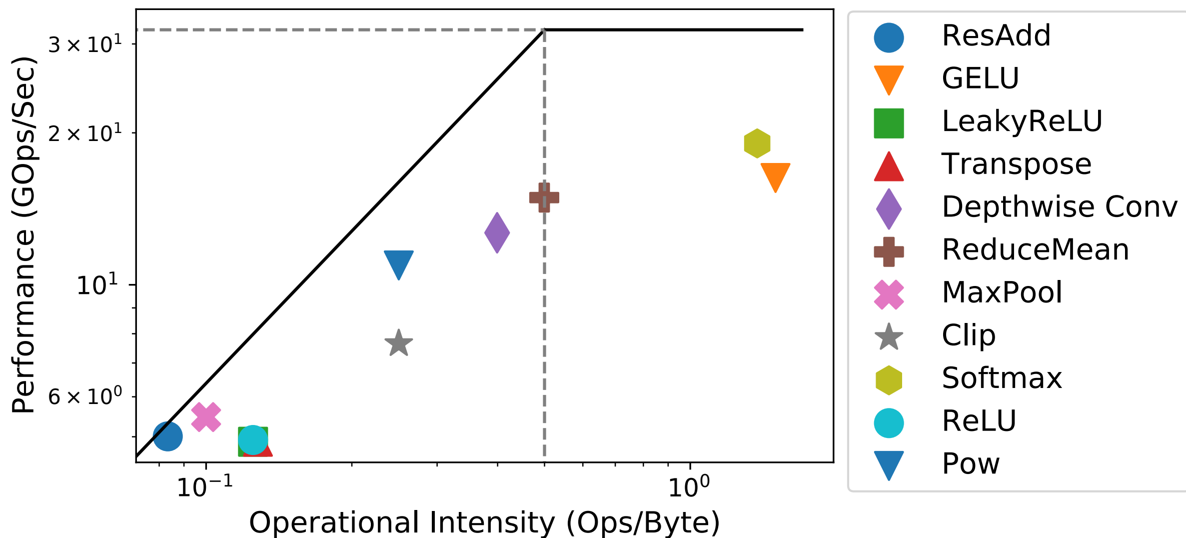


Figure 4.4. Roofline model for a number of prevalent non-GEMM operators.

low computational intensity and data reuse. Figure 4.4 shows a roofline [270][3] analysis for a set of prevalent non-GEMM operators. As shown, most of the analyzed operators (other than Softmax and GeLU) fall within the memory-bound region of the roofline model. This is in contrast to Conv/GEMM operations that are generally computation-bound [275]. This distinction necessitates architecture design considerations.

4.2.2 Requirements for Non-GEMM Execution

Inspired by the above characteristics, below we list three key requirements to efficiently execute non-GEMM operations.

R1: In-tandem execution of GEMMs and non-GEMMs. To reduce the data exchange among GEMM and non-GEMM units through off-chip memory, prior work [53, 181] suggests "Layer Fusion". To leverage layer fusion, the intermediate activations ought to be communicated between GEMM and non-GEMM units via on-chip memory subsystem for a sequence of fused layers. However, this data communication at the granularity of entire layer outputs is neither trivial nor efficient, due to following reasons: (1) The limited on-chip memory of the DNN

[3]We performed the experiments on Tandem Processor with the configuration shown in Table 6.1 of Section 4.6.1.

accelerators may not be sufficient to keep the entire intermediate activations on the chip (e.g., the intermediate activation footprints for a DNN like VGG-16 reaches beyond 6 MBytes). (2) This hinders the execution overlap of the fused layers and keeping both units busy at the same time. As such it results in underutilization of GEMM and non-GEMM units. In essence, the data transfer ought to be performed at a finer granularity, i.e., tile granularity (partial output tensors). *This fine granularity of coordination requires the non-GEMM unit to seamlessly work in tandem with the GEMM unit, while retaining minimal data transfer and reformatting overhead.*

R2: Balanced efficiency and programmability in designing non-GEMM unit. The diversity of the non-GEMM operators calls for a degree of programmability in their processing unit. Nonetheless, this programmability should not emerge at the cost of noticeable efficiency reductions. This is particularly important because the inefficiency of the non-GEMM unit can potentially make it the performance bottleneck and result in stalling the GEMM unit as well. Therefore, it is crucial to strike a balance between programmability and specialization in the design of the non-GEMM unit.

R3: Orchestrating the execution across non-GEMM and GEMM units. Having both GEMM and non-GEMM acceleration units in one coherent system requires adequate support for execution orchestration between these units. In particular, (1) the DNN nodes need to be effectively dispatched to their pertinent processing units, (2) GEMM and non-GEMM units need to diligently synchronize and handshake together at the right time to realize in tandem execution and back-and-forth interactions.

4.2.3 Spectrum of Approaches to Support Non-GEMM Layers

Table 4.2 lists prior methods and analyzes them with respect to: (1) in tandem execution with the GEMM unit, (2) programmability to support the wide range of non-GEMM operators in modern DNNs, (3) specialization of non-GEMM unit for effecient execution, and (4) the capability to control and orchestrate the end-to-end DNN execution. In addition to the qualitative discussions in this section, Section 5.6.2 provides quantitative comparisons.

Table 4.2. Comparison of prior approaches for supporting non-GEMM operators with this work. † indicates that these aspects are supported partially.

Design classes	Working in tandem with GEMM Unit	Specialization	Programmability	Execution Control
Offchip CPU fallback	✗	✗	✓	✓
Dedicated on-chip hardware units	✓	✓	✗	✗
Onchip RISC-V core (+ dedicated units)	✗†	✗†	✓	✓
General purpose vector unit	✓	✗	✓	✗
This work (Tandem Processor)	✓	✓	✓	✓

Class (1): Off-chip CPU fallback. A swath of neural network accelerators [55, 92, 291, 29, 28, 185, 169, 111, 74, 57, 94, 219, 293, 191, 276, 232] presume an off-chip CPU to which they fall back for non-GEMM operations. Additionally, the off-chip CPU is in charge of orchestrating the end-to-end DNN execution. While this approach yields ultimate programmability, it also impedes performance. First, off-chip CPU falls short in working in tandem with the GEMM unit. This is mainly because of the nontrivial overhead of back-and-forth data transfer through communication channels with modest bandwidth (e.g. PCIe). Furthermore, this back-and-forth communication often bears additional data conversions (e.g., integer to float and vice versa). This loosely-coupled design and integration hinder the support of execution optimizations such as software pipelining. Second, this approach does not offer specialized execution for non-GEMM operations, hence, naturally less efficient.

Class (2): Dedicated on-chip hardware units. An alternative strategy [58, 106, 215, 59, 158, 221, 129, 184, 79, 147, 15, 80, 90, 222, 167, 112, 178, 27, 39, 229, 31, 216, 145, 97] is to equip the GEMM accelerator with a set of dedicated units, each of which is exclusively customized for a specific type of non-GEMM operation. Generally, these dedicated units are tightly integrated with the GEMM unit (can indeed work in tandem), but do not offer execution orchestration and naturally operate as the slave of the GEMM unit. The inherent nature of "dedicated" hardware units warrants less scalability and extensibility for the following reasons: First, it is not scalable

to augment neural accelerators with dedicated units for each single type of non-GEMM operation. Second, this prohibits the accelerator to support brand-new non-GEMM operations as a result of evolving DNNs. Due to this limitation, in the case of unsupported operations these accelerators *must* fall back to an off-chip CPU, resembling the first execution class (See Table 4.2).

Class (3): Using an on-chip RISC-V core. Recent inspiring accelerators [95, 239] integrate an on-chip RISC-V CPU core with a GEMM unit. The on-chip core executes the non-GEMM operators as well as controls on-chip resources. Gemmini [95] extends the RISC-V ISA with a set of specialized instructions and dedicated hardware units to exclusively accelerate a set of non-GEMM layers. Although this class obviates off-chip CPU communication in the first class, it only partially enables in tandem execution between two isolated acceleration units (GEMM unit and the RISC-V core). The nontrivial overhead of data conversion and/or tensor layout transformation still persist. Furthermore, optimization techniques such as software pipelinig for non-GEMM operations (those supported on RISC-V core) can not be maintained. As the recent detailed study by Meta [239] delineates, such overheads are quite substantial and can result up to 60$\times$ to 88$\times$ runtime overhead for non-GEMM layers compared to their preceding GEMM layer.

Class (4): On-chip general-purpose vector unit. Another class of design is to employ on-chip general-purpose vector units. Nvidia Streaming Multiprocessor (SM) units [182] that consist of tensor cores (GEMM units) and CUDA cores (general-purpose vector units) belong to this design class. Another examples are Google TPU [128, 127], Tesla Dojo [247], and Brainwave [90] that encompass either SIMD or vector units for non-GEMM execution. Vectorized execution leverages the inherent parallelism in non-GEMM layers for increased performance improvement. Additionally, these vector units often work in tandem with the GEMM units. However, these units do not handle the execution control and fall short in specialization.

Tandem Processor class. To address all the three requirements discussed in Section 4.2.2, this paper offers the design of a SIMD processor that is companion to the GEMM unit. This processr can effectively operate in tandem with the GEMM unit, while striking a balance between

customization and programmability for non-GEMM operations. Additionally, the proposed processor is in charge of orchestrating the end-to-end DNN execution and hence eliminating the need for an additional general-purpose companion CPU.

4.3 Design Considerations for Tandem Processor

4.3.1 Memory Subsystem Design

The low computational intensity and at the same time the sizable tensor operands for non-GEMM operators prompt the memory subsystem to repeatedly *stream* data from off-chip memory. Thus, a locality-oriented hierarchical memory sub-system (i.e., vector register file and cache(s)) and conventional load/store data communication, necessitates an excessive number of memory instructions to deliver off-chip data to/from vector register files, funneling through the memory hierarchy. To address this, we use the following insight: *Non-GEMM layers most often operate on statically-structured tensor operands with a-priori known dimensions in a streaming fashion.* Tandem Processor replaces the vector register file and cache hierarchy with a collection of single-level software-managed on-chip scratchpad memories to store tensors and immediate values. To manage data movements between off-chip/on-chip memories, we design a Data Access Engine microarchitectural unit. This unit can be configured and invoked by few explicit load/store memory instructions per tile to fetch entire tenors. Such data movement merely appears at the boundary of a tile, blocking any further intervention from the off-chip memory subsystem. This design choice amortizes the von Neumann overhead of frequent memory instructions over a large number of compute operations (i.e., a tile).

4.3.2 Specialized On-Chip Data Access Mechanism

While using large on-chip scratchpads bear its benefits, encoding the scratchpad addresses presents new design challenges. The scratchpad addresses do not readily fit in an Instruction Word as opposed to IDs of vector register files. In addition, calculating on-chip scratchpad addresses

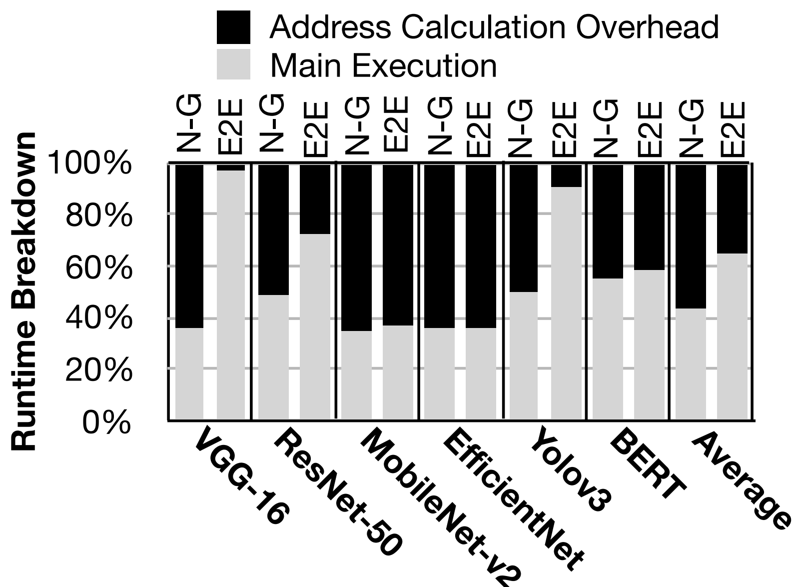


Figure 4.5. Overhead of address calculation using arithmetic instructions. "N-G" and "E2E" denote the runtime for Non-GEMM and End-to-End execution. This experiment was performed on Tandem Processor with Table 6.1 configurations and all design specializations except for on-chip data access mechanism.

requires excessive number of arithmetic instructions. For instance, per two-operand arithmetic/-logic instruction, three extra instructions would be required solely for address calculation. As Figure 4.5 shows, this address calculation would impose runtime overhead. On average, 56% of the runtime for non-GEMM layers and 35% of end-to-end DNN runtime would be spent only on address calculation. To tackle this challenge, *we disentangle the address calculation and compute operations via pipeline parallelism.* This pipeline parallelization relieves the burden of address calculation from compute units and cancels out its runtime overhead.

To enable this design optimization, we leverage the regularity of tensors for non-GEMM operands and use a collection of statically-known strided accesses to fetch the tiled data. In another word, walking over each dimension of tensor operands can be regulated by a tuple of $\langle$Offset, Stride $\rangle$. Hence, if these tuples can be embedded in a single instruction along with compute operations, upon being inferred at the decode stage, the scratchpad addresses can be calculated in parallel with compute operations. Even with this design optimization, providing three such tuples for a non-GEMM layer would be unrealizable in a limited-width instruction

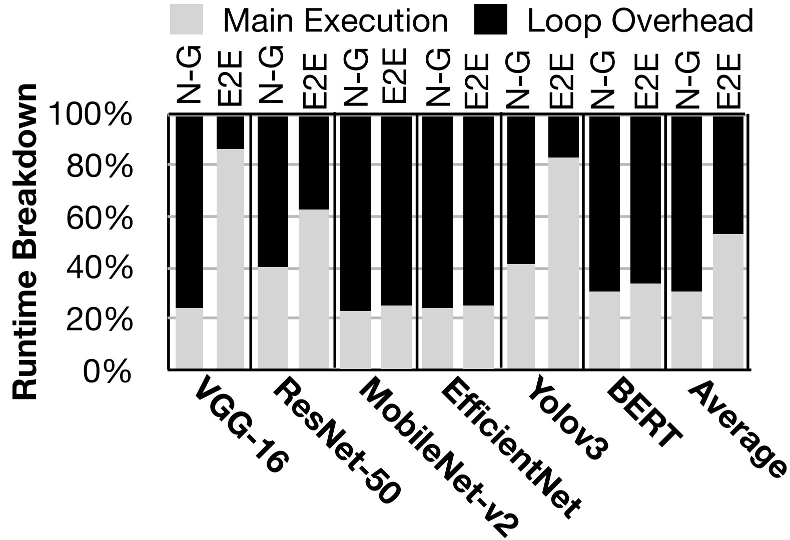


Figure 4.6. Runtime overhead of loop execution using branch logic across benchmarks. "N-G" and "E2E" denote the runtime for Non-GEMM and End-to-End execution.

word. Instead, we forge scratchpad accesses through indirect strided address calculations. We formulate these strided accesses using ⟨Scratchpad ID, Iterator Index ⟩ format. The Scratchpad ID designates the scratchpad tier and the Iterator Index points to an entry in an Iterator Table. Each entry in the Iterator Table stores a tuple of ⟨Offset, Stride ⟩ for each operand. Following this mechanism, Tandem Processor indirectly calculates a target scratchpad address by adding Offset and Stride values. The use of this concerted operand encoding scheme and software-controlled iterators reduce the code footprint and its prompted runtime overheads. In addition, this design optimization realizes the embedding of strided addresses and compute operations into a single 32-bit instruction word (See Section 4.4.3). With this mechanism Tandem Processor supports address calculation as well as compute operation on the same pipeline path with shared control. This is in contrast to prior work [189, 280] which supports decoupled access/execute engines for address generation.

4.3.3 Specialized Loop Execution

Non-GEMM layers have well-formed nested loops of primitive operations with pre-determined iteration counts. Using conventional general-purpose loop execution model (e.g.

Table 4.3. **Non-GEMM examples and their implementations using primitives.**

Non-GEMM Layers	Primitive Operations
LeakyRelu	Max, Mul
Clip	Max, Min
GeLU [137]	5×Mul, 3×Add, Sign, Abs, Min
Sqrt [137]	4×Add, 4×Ceil, 4×Shift, 3×Div, Mul
Softmax [137]	4×Add, 4×Mul, Floor, Max, Sub

conditional branch, rollback units) incurs significant runtime overhead. To better understand this overhead, we use Tandem Processor + GEMM unit with all specialization enabled, except loop execution. Figure 4.6 shows, on average, conventional loop logic incurs 70% and 47% runtime overhead for non-GEMM layers and end-to-end DNN execution, respectively. To alleviate this overhead, we can leverage the regularity of the loop constructs in non-GEMM layers and devise specialized loop execution semantics.

To that end, Tandem Processor uses software-managed tables in the fetch pipeline stage to orchestrate the execution of nested loop constructs in hardware. Prior to execution, these tables are configured *once* with the iteration counts and corresponding number of nested loop levels. Once configured, these specialized tables are used repeatedly to execute the loop body until the termination criteria is reached (e.g. maximum number of iterations).

4.3.4 Arithmetic Logic Units Design

Tandem Processor ALU operations. To support the execution of diverse set of non-GEMM layers (Figure 4.2), one approach would be to use dedicated specialized instruction for each layer. However, this would lead to a design similar tp the second class in Section 4.2.3. We instead use an alternative approach and leverage the feasibility of implementing non-GEMM layers with a set of simple primitive operations [137, 24]. Table 4.3 lists sample of non-GEMM layers and their corresponding primitive operations. Therefore, to support sufficient programmability, Tandem Processor advocates for only implementing a set of primitive operations. We consider a union set of these primitives, which is comprehensive enough to support non-GEMM layers shown in

Table 4.1. With this design, Tandem Processor offers better hardware resource utilization and reuse across a larger set of operations. This is in contrast to dedicated specialized units which may be used less frequently.

Tandem Processor ALU precision and datatype. Prior works have shown that integer-only arithmetic can be used for inference execution of CNNs [121, 278] and transformers [137] with virtually no repercussions on accuracy. In addition, while GEMM layers and few non-GEMM layers such as Relu and Clip are amenable for low-precision INT8 implementation [121], some non-GEMM layers such as ResAdd, GeLU, Softmax require INT32 precision [278, 137]. To provide sufficient mathematical precision for all non-GEMM operators, we design ALUs with INT32 precision in Tandem Processor. As a complementary benefit of this design, we do not need to perform additional data casting from GEMM to non-GEMM unit. This is because GEMM units typically accumulate the partial results in INT32 precision [128, 127, 137, 55, 92, 222]. In contrary, since GEMM layers may use lower precision, a datatype casting instruction is required when activations move from non-GEMM to GEMM unit. These instructions enable conversion to lower precision integer values (e.g. INT4, INT8, INT16, etc.).

4.3.5 Tandem Processor Integration with GEMM Unit

Below, we discuss three design aspects to seamlessly integrate the Tandem Processor with the GEMM unit.

Coordination granularity. We use tile granularity for software pipelining to facilitate execution overlap between GEMM and non-GEMM units and improve resource utilization. In addition, tile-based execution better conforms with limited on-chip memory. As Figure 4.7 shows, the in tandem coordination of GEMM unit and Tandem Processor at tile granularity increases the compute resource utilization by 26% and 17% for GEMM unit and Tandem Processor, respectively, compared to layer level granularity. Note that an operand-level granularity is less efficient. This is because some non-GEMM operators, such as depthwise convolution and global average pooling, require arbitrary intra-tensor accesses to GEMM outputs for consecutive operations.

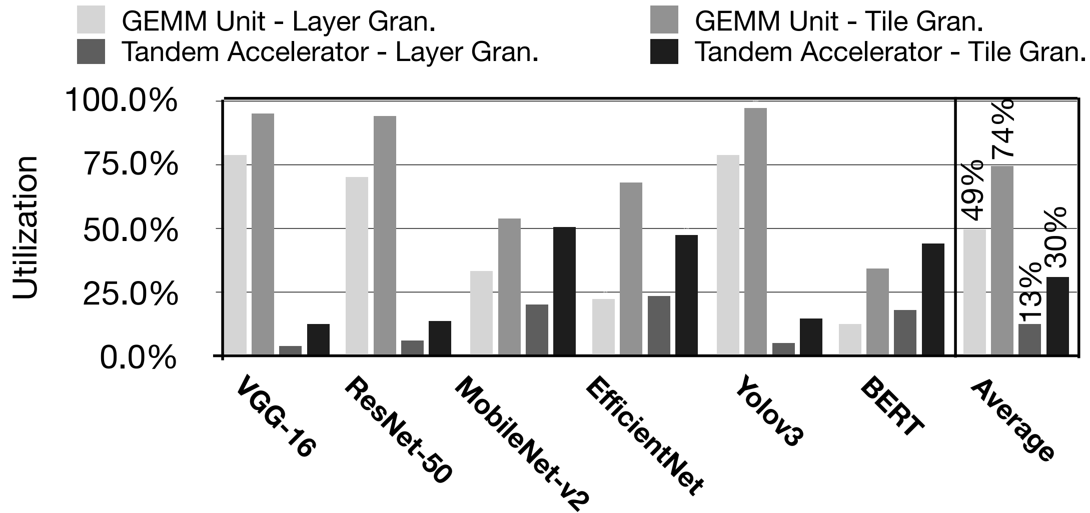


Figure 4.7. Compute resource utilization for GEMM unit and Tandem Processor for layer level and tile level granularity of coordination. For layer granularity all the Tandem Processor's specializations are enabled except the tile level coordination.

This arbitrary access pattern results in frequent stalls, curtailing the overall performance.

Communication mechanism. To enable tile-based coordination, one probable approach is to directly move/copy tiled data from GEMM unit's Output BUF to Tandem Processor's private scratchpads. However, this design decision incurs communication overhead at the boundary of each accelerator units, requiring complex coordination mechanism. Alternatively, we employ a concerted on-chip memory shared and co-managed by the individual accelerator units. To implement this, we enable a fluid ownership of the GEMM unit's Output BUF for Tandem Processor, obviating redundant data communications. After the GEMM unit completes storing the intermediate data in the Output BUF, Tandem Processor takes the ownership of the buffer and directly execute its computations on the stored data.

Synchronization mechanism. To support in tandem execution and enable a fluid ownership mechanism for on-chip buffers, it is imperative to devise a synchronization mechanism between acceleration units. To simplify hardware, we leverage the regularity in the execution pattern of DNNs and advocate for a *software-controlled* approach. We delegate this responsibility to compiler to weave synchronization instructions (See Section 4.4.3) between GEMM and

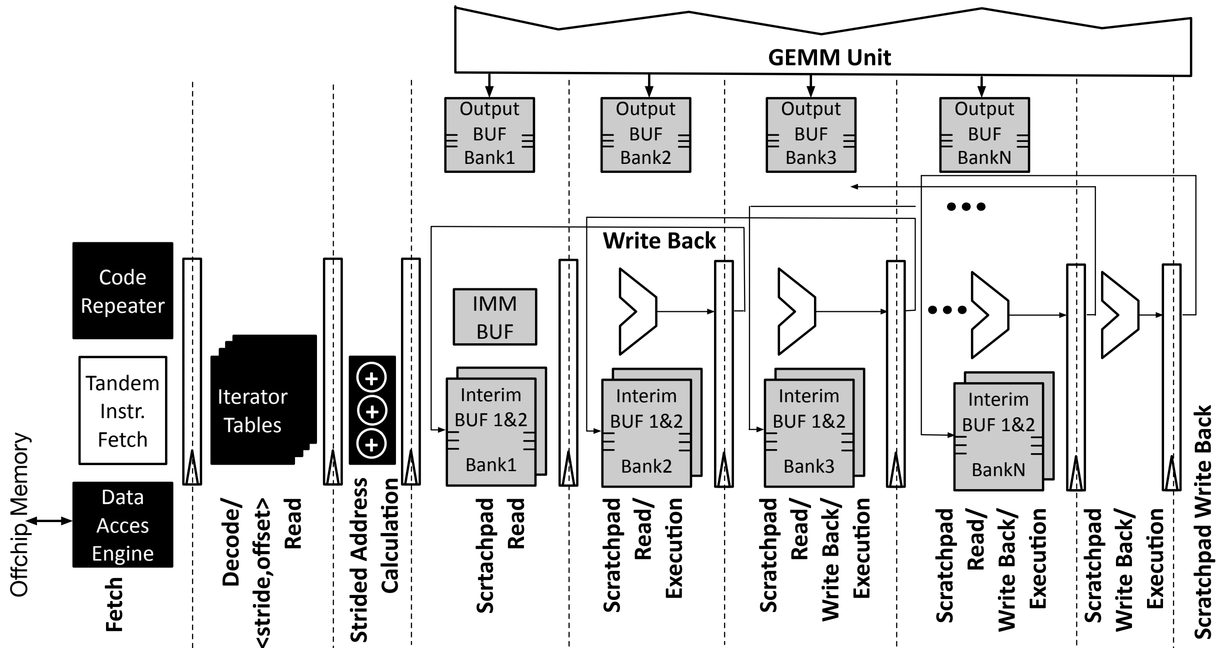


Figure 4.8. The Tandem Processor pipeline microarchitecture.

non-GEMM instructions. These synchronization instructions realize the following objectives: (1) They identify the code regions for GEMM unit and Tandem Processor, facilitating the instruction dispatch. (2) They define the flow of execution between GEMM and non-GEMM units. (3) They govern the handshaking mechanism between the acceleration units. For instance, enforcing the release of ownership of the Output BUF after Tandem Processor completes the execution.

4.4 Microarchitecture and ISA for Tandem Processor

4.4.1 Tandem Processor Pipeline Microarchitecture

In this section, we discuss the major aspects of the Tandem Processor's pipeline microarchitecture, illustrated in Figure 4.8.

On-chip memory organization. We refer to Tandem Processor single-level scratchpads as Namespaces, which are shown with gray colour in Figure 4.8. Interim BUF 1&2 namespaces represent the central Tandem Processor's on-chip scratchpads that operate as a storage medium for tensor operands as well as their intermediate results. To use single-port SRAMs and reduce

their energy/area cost, two Interim BUFs are used to facilitate accessing two operands from scratchpads simultaneously. These scratchpads, which bridge the off-chip memory and Tandem Processor, are populated/drained by a Data Access Engine at a tile granularity. The Tandem Processor compiler configures the Data Access Engine by setting the base address of the off-chip source along with a series of stride values. Note that, the tiled data may be even dispersed across non-contiguous regions of memory lines, yet statically arranged in strided patterns. This setup along with the statically optimized data layout enables seamless walk through regions of memory in fixed-sized steps to fetch a collection of memory lines funneled into one of the Interim BUF s. Note that, Tandem Processor does *not* support stand-alone non-tiled load/store instructions. IMM BUF namespace serves as a small 32-slot scratchpad for immediate values in non-GEMM operations. This buffer is programmed with a series of customized instructions at the onset of non-GEMM layer execution. The last namespace is Output BUF, which serves as the GEMM Unit's buffer for output values.

Specialized on-chip data access. We place the Iterator Tables that are used to store the offset and stride information for scratchpad accesses at the decode stage of the Tandem Processor pipeline. There is a dedicated Iterator Table for each namespaces of the Tandem Processor. Upon decoding one arithmetic/logic instruction, the ⟨Namespace ID, Iterator Index ⟩ retrieves the address calculation information from the corresponding Iterator Table. The resulting outputs of accessing the Iterator Table s is a triplet address, two for source operands and one for destination operand. Each element of the triplet is a tuple of ⟨offset, stride ⟩, indicating that target data resides in Scratchpad[offset + stride]. The triplet address is passed down to the subsequent pipeline stage (Strided Address Calculation) that repetitively assembles a series of scratchpad addresses, each as the result of offset + stride computation. The scratchpad indices propagate down the multi-staged execution pipeline to fetch the tiled operands, perform the non-GEMM operations, and write back the resulting data to the pipeline back-end.

Nested loop support. To realize the nested loop execution, the Code Repeater module uses three

tables: A table stores the compiler-defined iteration counts. Each entry of this table maintains the configuration of one of the loop nesting levels. The compiler organizes the loop configuration instructions from the outermost loop to the innermost one. At the Decode/<Stride,Offset>/Read pipeline stage, Code Repeater stores the number of iterations in each table entry, which is indexed using a pointer that keeps track of the number of nested loops. Once the Code Repeater is configured, it uses the second table with similar structure of entries to keep track of the current iteration of the loops. Whenever, the Code Repeater exhausts the iterations of a loop level, it decrements the pointer to update the iterations of the ensuing outer loop. Finally, the Code Repeater uses a collection of identical tables that store the information about what Iterator ID s need to be exercised for each operand at a certain loop level.

Considerations for pipeline frequency. The on-chip scratchpads in Tandem Processor are relatively large SRAMs that may not be placed close to the ALUs during floorplaning. This implicates that direct accesses to on-chip scratchpads may result in timing repercussions due to probable long wire lengths. In addition, simultaneously accessing the SRAM banks mandates the design to support high fanout memory address ports. To that end, we supplement the design with an additional degree of pipelining along the ALU lanes to interleave the scratchpads read/write stages with the execution ones, while dispersing ALUs in different pipeline stages. Tandem Processor forwards the read/write addresses and their germane control signals (e.g. type of ALU operation) to scratchpad banks along the ALU lanes in a pipelined fashion.

4.4.2 Synchronization Logic and Overall Execution Flow

To execute DNNs, we consider a block-structured execution semantic for the DNN accelerator that utilizes Tandem Processor (NPU-Tandem). A block can be one of the followings: (1) a single GEMM layer, (2) a group of bundled non-GEMM layers, (3) a GEMM layer followed by a group of bundled non-GEMM layers. To realize the in tandem execution of GEMM unit and Tandem Processor, a uniform tiling scheme is required across the fused layers in one block. Figure 4.9 illustrates the high level view of the execution controller logic for Tandem Processor,

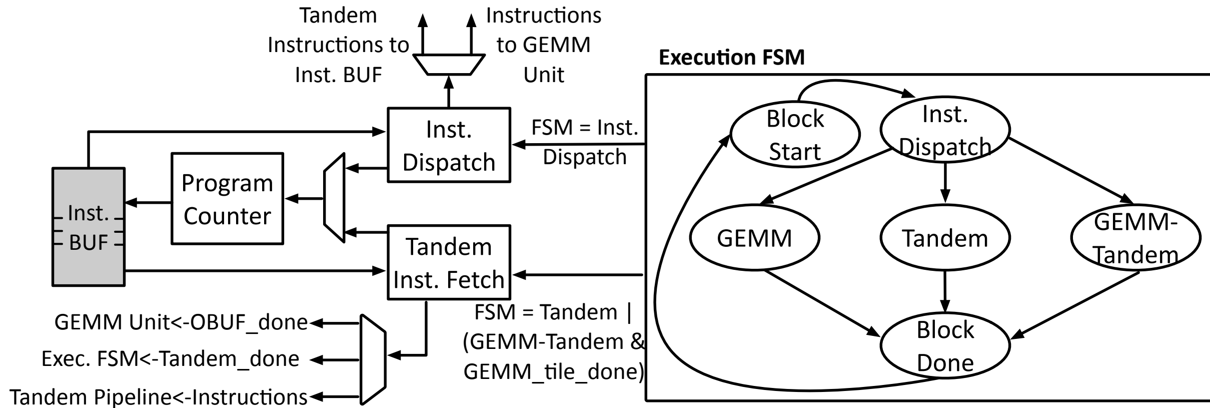


Figure 4.9. The execution controller.

which orchestrates the end-to-end DNN execution and synchronizes the GEMM unit and Tandem Processor. Below, we describe the main states of its execution FSM.

Block start and instruction dispatch states. Once all the instructions of a block are available on Tandem Processor's Inst. BUF, the execution of a block starts and the FSM switches to the Inst. Dispatch state. At this state, the Tandem Processor's Inst. Dispatch unit drives the Program Counter to walk over all the instructions of a block. The Inst. Dispatch decodes the synchronization instructions that are used to mark the boundaries of GEMM and non-GEMM instructions (see Section 4.4.3). Consequently, the unit sends the GEMM instructions to the GEMM unit, while writing back the non-GEMM instructions to the Inst. BUF for their execution. After the dispatch is done, based on the structure of the program block, the execution FSM switches to either of these three states: GEMM state, Tandem Processor state, and GEMM-Tandem Processor state. Below, we discuss the last two states which are more relevant to this work.

Tandem Processor state. This state is used when an instruction block comprises only non-GEMM layers. At this state, the execution FSM triggers the the Tandem Inst. Fetch to fetch the non-GEMM instructions from the Inst. BUF and forward them to Tandem Processor pipeline. Once Tandem Processor completes executing all the instructions, the Tandem Inst. Fetch unit sends a handshaking signal to the execution FSM logic. The execution FSM loops back to this state if there are remaining tiles and the Tandem Inst. Fetch starts executing the instructions from

the beginning for the next tile. To ensure the off-chip memory access instructions are updated for different tiles, the first tile is used to initialize configurations for the Data Access Engine. For rest of the tiles, the Data Access Engine reuses the initialized configurations and incrementally updates them based on the current tile.

Tandem Processor-GEMM unit state. If the execution block is formed of a GEMM layer followed by a series of non-GEMM layers, the execution FSM transitions to GEMM-Tandem Processor state after the instruction dispatch. Whenever the GEMM unit finishes a tile, it releases the Output BUF and sends a handshaking signal to Tandem Processor. If Tandem Processor is idle, the Tandem Inst. Fetch gets triggered. Utilizing the double-buffering scheme, the GEMM unit proceeds to the next tile, while Tandem Processor takes the outputs of the GEMM-completed tile and performs the non-GEMM operations. To avoid stalls in the GEMM unit caused by the Output BUF being occupied by Tandem Processor, the compiler inserts a synchronization instruction (see Section 4.4.3) right after the instructions consuming the data on the Output BUF. At this time, the Tandem Inst. Fetch sends a handshaking signal to the GEMM unit and Tandem Processor releases the Output BUF. Once Tandem Processor finishes a tile, it uses the synchronization instruction that marks the end of the non-GEMM program to alert the execution FSM. The execution FSM puts the Tandem Processor in the idle state until the next tile from GEMM unit becomes available. Once all the tiles are done, the execution FSM transitions to the Block Done state.

4.4.3 Tandem Processor ISA

We devise the ISA for Tandem Processor with following considerations: (1) providing synchronization semantics to realize the in tandem execution of Tandem Processor and GEMM unit, (2) efficiently encoding customized on-chip memory accesses, (3) minimizing repetitive instructions through customized loop constructs, and (4) supporting a comprehensive set of primitive operations. Figure 4.10 summarizes the instruction formats for Tandem Processor ISA. Below we discuss its instruction classes in more details.

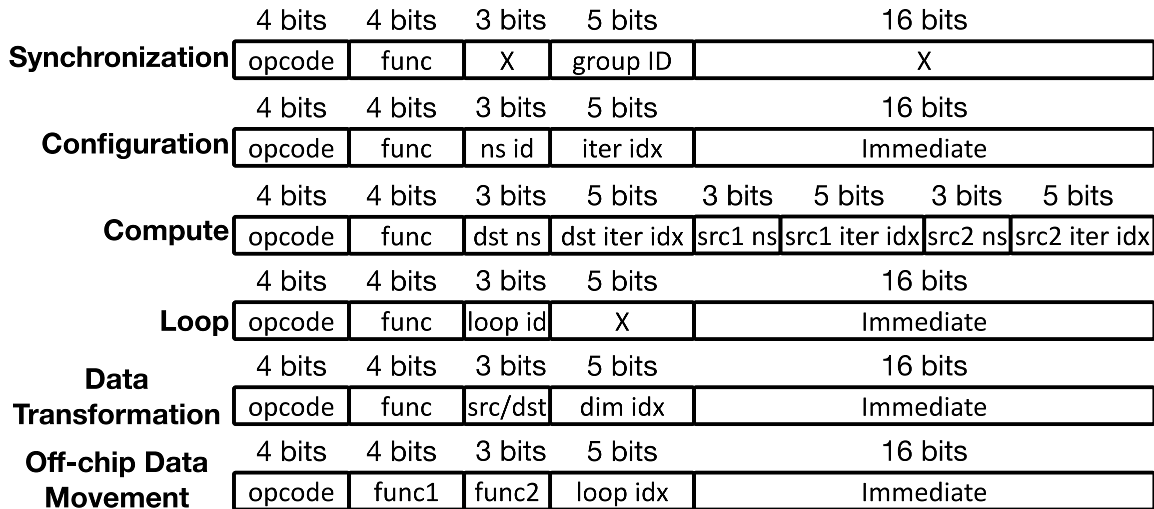


Figure 4.10. Tandem Processor instruction set formats that does not use any registers.

Synchronization instructions. This instruction helps Tandem Processor and GEMM Unit coordinate in a lock-step. As the func bits, the START/END along with EXEC bit identifies the regions of instructions that belong to Tandem Processor and GEMM Unit, which helps dispatch instructions to the appropriate unit. Also, this instruction can be used with EXEC bit to notify the GEMM Unit that the execution of non-GEMM operations of the running tile is completed, or with BUF bit to notify GEMM Unit that the OUTPUT BUF is released and can be used for the execution of subsequent tile.

Configuration instructions. There are two opcodes for configuration instructions. ITERATOR_CONFIG is used to set the BASE_ADDR and STRIDE for the scratchpad address calculation, while the ns id and iter idx fields identify the target namespace and the index to its corresponding Iterator Table. Also, this instruction is used to set the immediate values in the IMM BUF. The DATATYPE_CONFIG opcode configures the ALU with the precision of source and destination operands for datatype cast operations.

Compute instructions. This class dictates the computation on Tandem Processor to support the primitive operations for diverse set of non-GEMM operations. Opcode ALU supports Add, Sub, Mul, MACC, Div, Max, Min, Shift, Not, AND, and OR. This also includes MOVE

and COND_MOVE instructions to support scatter/gather operations or concatenation. Opcode CALCULUS consists of common mathematical operations such as absolute value and sign. Opcode COMPARISON supports logical compare operations. The operands for each instruction are specified by using a 3-bit ns id to locate the buffer, and a 5-bit iter idx corresponding to the stride and offset.

Loop instructions. To configure the Code Repeater, the LOOP opcode is used with SET_ITER function bits to specify the iterations for each loop identified by loop id. The SET_NUM_INST function is used to identify the number of instructions in the loop body. To cope with the customized on-chip memory accesses for each loop dimension, the SET_INDEX function is used, while the rest of the instruction bits are used to set the associated ⟨ns ID, iter idx⟩ for the three operands (similar to compute instructions). The loop instructions are designed to support arbitrary levels of nesting (up to eight, each of which is identified by loop id field) needed by non-GEMM operators.

Data transformation instructions. Tandem Processor ISA includes a set of instructions to permute multi-dimensional tensors. SET_BASE_ADDR / SET_LOOP_ITER / SET_LOOP_STRIDE functions configure the base addresses, shapes, and strides, respectively, for both the source and destination's tensor dimensions (identified by dim idx). Tandem Processor ISA also includes a DATATYPE_CAST opcode that casts tensor elements to various fixed-point representations such as FXP32, FXP16, FXP8, and FXP4 needed by the GEMM unit.

Off-chip data movement instructions. TILE_LD_ST opcode describes the data tile transfer between off-chip memory and Interim BUFs. LD/ST_CONFIG_BASE_ADDR function is used to generate the base addresses of each tile, then the shape and strides are configured using LD/ST_CONFIG_BASE_LOOP_ITER/STRIDE. LD/ST_CONFIG_TILE_LOOP_ITER/STRIDE instructions are used to configure the Data Access Engine to generate the addresses required for each tile. Finally, LD/ST_START instruction along with source/destination namespace, data width, and size triggers the Data Access Engine.

4.5 Compiler Support for Tandem Processor

4.5.1 Optimizations

Tiling. One key variable for tiling is its size: tile must be big enough to encompass all the adjacent elements of an input tensor for the non-GEMM operation while small enough to fit on the limited on-chip scratchpads. For instance, to perform Depth-wise Conv operation with a kernel size 5×5, it would require Tandem Processor to have access to all the elements in the 5×5 patch or it is inevitable to stall. Another important consideration is the dimensions to be tiled. For example, the reduction dimensions in GEMM operations should not be tiled as tiling the reduction dimension would make GEMM Unit produce partial results that would be insufficient for the Tandem Processor to perform its operations, causing it to stall.

Dependency relaxation. Tandem Processor leverages the regularity in the non-GEMM operations and eliminates the dependency check in the hardware to simplify it, while shifting the burden to the compiler. However, existence of sequence of instructions in loop bodies dictates dependencies among instructions. Tandem Processor compiler leverages *loop fission* [33] to remove dependencies among series of instructions. Additionally, some non-GEMM operations such as MaxPool has a long sequence of dependencies among instructions. For example, MaxPool with 3×3 kernel requires 9 MAX instructions with dependencies. For such cases, Tandem Processor compiler leverages *loop interchange* [33] to relax the dependencies (e.g., placing kernel height and width loops to be the outermost ones).

4.5.2 Compilation Workflow

Figure 4.11 describes the compilation workflow for Tandem Processor. The compiler uses the ONNX representation model of DNNs and the architecture configuration of Tandem Processor (e.g. number of lanes, Interim BUF) as its inputs. The compiler maps the ONNX node to pre-defined operation templates. However, as discussed in Section 4.3.4, not all non-GEMM operators are directly supported by Tandem Processor. Therefore, for such complex operations

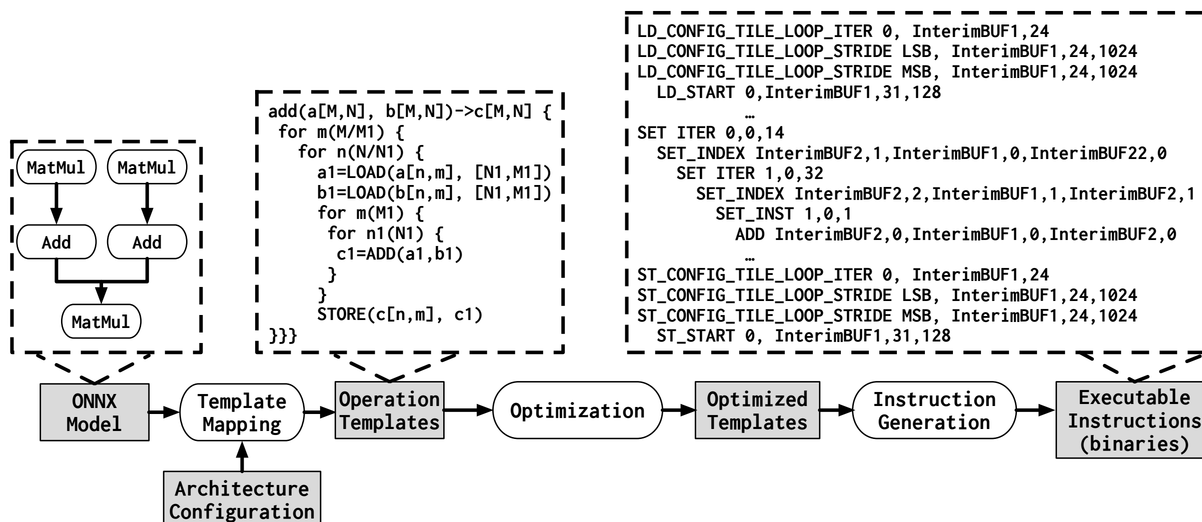


Figure 4.11. Compilation workflow.

(e.g., Softmax, Sqrt, Gelu) the compiler translates them to an integer-based counterpart [24, 137]. After mapping to the templates, the parameters of the operation templates are replaced with real values according to the ONNX layers. The compiler then performs the aforementioned optimizations. Finally, the compiler iterates the statements in the template and lowers them into instructions based on the Tandem Processor ISA.

The memory allocation statements are lowered to TILE_LD_ST instructions. LOAD and STORE statements are lowered to TILE_LD_ST with BASE_LOOP_ITER/STRIDE functions for each of the LOOP variables, to set the number of iterations and strides in DRAM (a summarized version is shown in Figure 4.11). After loops are configured, the tile transfer instructions (LD/ST_CONFIG_TILE_LOOP_ITER/STRIDE) can be generated using the tile shape information in the LOAD and STORE statements. Each compute operation such as ADD are individually lowered to a set of inner LOOP instructions along with the corresponding compute instruction (a summarized version is also shown). For fused operations with compute operation reading from Output BUF, the compiler generates additional synchronization instructions.

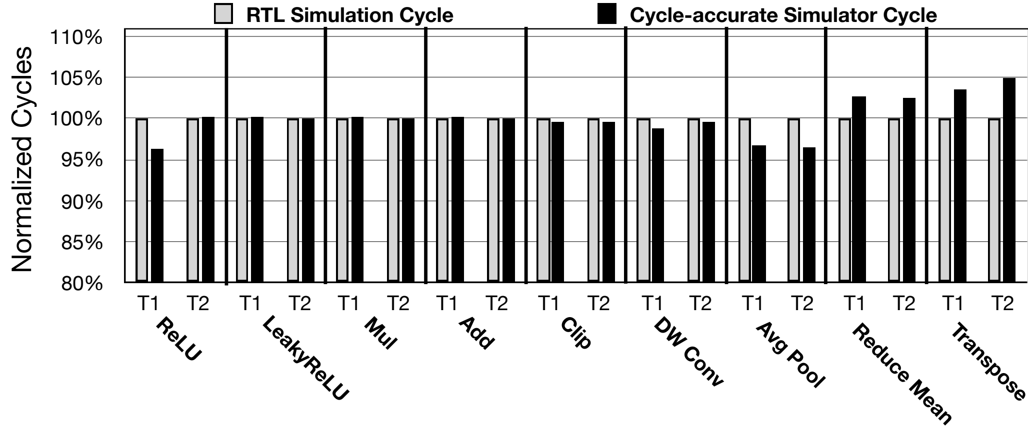


Figure 4.12. Accuracy of cycle-accurate simulator (normalized to RTL simulation).

4.6 Evaluation

4.6.1 Methodology

Benchmarks. We evaluate DNNs from image classification (VGG-16 [224], ResNet-50 [110], MobileNetv2 [212], EfficientNet [244]), object detection (Yolov3 [199]), and natural language processing (BERT [78]). These DNNs consitute a diverse set of layers with various dimensions and types of operations as shown in Table 4.1. We use batch size of 1 for evaluations, which is used for real-time AI [90], single-stream, and offline scenarios [197].

Hardware implementation and synthesis. We implement Tandem Processor in Verilog and synthesize it using Synopsys Design Compiler R-2020.09-SP4 with Global Foundries 65 nm standard cell library. We also perform floor planning and place and route using Synopsys IC Compiler L-2016.03-SP1. Additionally, to evaluate the design in a lower technology node, we synthesize Tandem Processor with FreePDK 15nm open cell library and meet the 1 Ghz target frequency. To obtain power of the design for comparison with baselines, we use the synthesis results in FreePDK 15 nm for logic cells and model the on-chip memory energy using CACTI-P [155].

Simulation infrastructure. We fully compile each DNNs in the form of ONNX to the Tandem Processor ISA and generate binaries. We develop a cycle-accurate simulator for Tandem Processor that uses compiler-generated instructions and provides cycle counts and energy statistics.

The simulator models each individual microarchitecture components of Tandem Processor. We validate the functionality of the simulator and RTL implementation by comparing the simulator/RTL-generated outputs with ground truth software implementation. Additionally, we compare the simulator cycle counts with RTL simulation and verify the fidelity of the performance results. Figure 4.12 compares the cycles measured by cycle-accurate simulation and RTL. For the sake of brevity only nine non-GEMM layers and two test cases per each is shown, while we performed the verification for the entire studied DNN layers and ten test cases per each. As shown, the cycles reported by cycle-accurate simulation follow the RTL cycle reports by an error margin of $\leq 5\%$. For end-to-end results, following the methodologies of [95, 209, 208, 222], we develop a cycle accurate simulator for a systolic array based GEMM Unit and integrate it with Tandem Processor simulation infrastructure following the insights in Section 4.3.5.

Comparison to off-chip CPU fallback and dedicated units (Class (1) and (2) in Section 4.2.3) We compare a DNN accelerator system composed of a GEMM Unit and Tandem Processor (NPU-Tandem) with the configurations listed in Table 6.1 to two baselines. As the first baseline, we consider a PCIe-attached (third generation with eight lanes) GEMM unit and an off-chip Intel Core i9-9980XE Extreme Edition CPU to support non-GEMM layers. As the second baseline, we augment the GEMM unit with a number of dedicated hardware units that support Relu, Clip, Residual Add, MaxPool, and scale & shift, similar to the design in [95]. This baseline still falls back to the CPU for unsupported layers. We measured the GEMM unit and dedicated units (for the second baseline) runtime using our aforementioned cycle-accurate simulator. The non-GEMM layers that needed to fall back to CPU were measured on Intel CPU using ONNX Runtime [77]. Finally, We measure the PCIe communication using a Xilinx Alveo u280 FPGA connected to host CPU via PCIe. We implement a dummy logic on FPGA to transfer the data over PCIe and measure its overhead for all required data transfers in benchmarks. All baselines use the same frequency, number of PEs, and on-chip memories as in Table 6.1. For energy comparisons, we estimate the power of GEMM unit using energy reports provided by prior

Table 4.4. Microarchitectural configurations of NPU-Tandem.

Configs/Units	Systolic Array	Tandem Processor
Dimensions	32x32	32 Lane
Scratchpads	384 KB	128 KB (Interim BUF 1&2)
Accumulators	128 KB	N/A
Datatypes	INT8 (Mult) and INT32 (Acc)	INT32
Frequency	1 GHz	1 GHz

works [84, 127], and model the energy of PCIe transactions according to another prior work [41].

Comparison to Gemmini [95] (Class (3) in Section 4.2.3). We compare NPU-Tandem with Gemmini [95] that integrates a systolic array, a set of peripheral dedicated units (similar to those mentioned above), and a RISC-V CPU core. We compile benchmarks using ONNX Runtime [77] to Gemmini and use cycle-accurate Firesim [131] simulator to obtain performance numbers. For a fair comparison, we exclude all the runtime/OS-related overheads. Additionally, to perform an iso-resource comparison, we use a scaled up Gemmini-like design that integrates the same number of Rocket cores as the number of ALU lanes in Tandem Processor. To obtain the results for this setting, we optimistically scale down the CPU runtime of Gemmini with the number of integrated Rocket cores.

Comparisons to general-purpose on-chip vector unit (Class (4) in Section 4.2.3). We also compare NPU-Tandem with two design points that integrate general-purpose vector units. First, we consider a hypothetical design point that integrates the same AVX-enabled Intel Core i9-9980XE Extreme Edition CPU with the GEMM unit. Second, we compare the NPU-Tandem to NVIDIA's Jetson Xavier NX GPU that also leverages specialized NVDLA [15] accelerator with INT8 execution. We run all DNNs on Xavier GPU using TensorRT v7.2.3 and use the average latency over ten runs.

4.6.2 Experimental Results

Comparison to offchip CPU fallback and dedicated units. Fig. 4.13 compares the performance of NPU-Tandem with baselines (1) using offchip CPU fallback and (2) using dedicated units.

The results are normalized to the baseline (1). On average, NPU-Tandem provides 4.0× and 2.9× speedup compared to baseline (1) and baseline (2), respectively. As shown in Figure 4.13, using dedicated units in baseline (2) leads to significant improvements over baseline (1) only for VGG-16 and ResNet-50, while the improvements are rather modest for other benchmarks. This is due to the lack of programmability and support for various and more modern non-GEMM layers, which makes the use of offchip CPU inevitable for the end-to-end execution. However, the seamless and tight integration of Tandem Processor with the GEMM unit brings forth sufficient programmability for the end-to-end execution and alleviates the need to use an offchip CPU. Tandem Processor not only eliminates the overheads of communication with offchip over PCIe and improving resource utilization, it also minimizes the overheads of instruction orchestration and data access compared to the general purpose CPU. The improvements provided by Tandem Processor are more pronounced for MobileNet-v2 (5.9× over baseline (1) and 5.4× over baseline (2)) and BERT (5.4× over baseline (1) and 4.5× over baseline (2)) due to the use of more complex non-GEMM operations in their structure (depth-wise convolution in MobileNet-v2 and large number of mathematical and transpose operations in BERT) that significantly affect the total runtime. However, for VGG-16, NPU-Tandem slightly underperforms baseline (2), due to the existence of large GEMM operations in this DNN and rather simple non-GEMM operations, for which the design of baseline (2) is heavily customized for. *The results in fact show that, as the DNNs evolve and use more complex structures, the benefits of Tandem Processor grow.*

Figure 4.14 compares the energy reduction offered by Tandem Processor with the other two baselines. On average, NPU-Tandem reduces the total energy consumption by 40.1× and 20.1× compared to baseline (1) and baseline (2), respectively. The results follow a trend similar to the performance comparisons. As the results show, the specialized low-power Tandem Processor fully replaces the power-hungry high-end CPU and reduces energy significantly.

Runtime breakdown. Figure 4.15 shows runtime breakdown for baselines (1),(2) and NPU-Tandem across DNN layers. For clarity, we highlight eight most common non-GEMM layers and

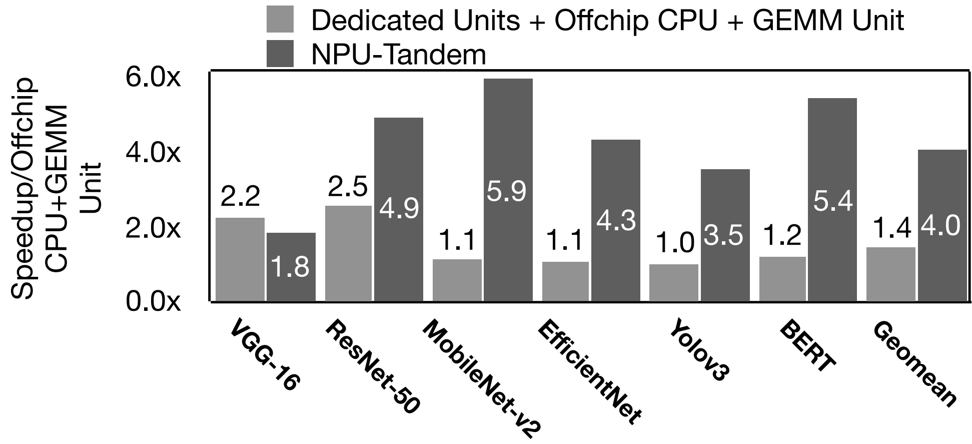


Figure 4.13. Performance comparison to offchip CPU fallback and dedicated units.

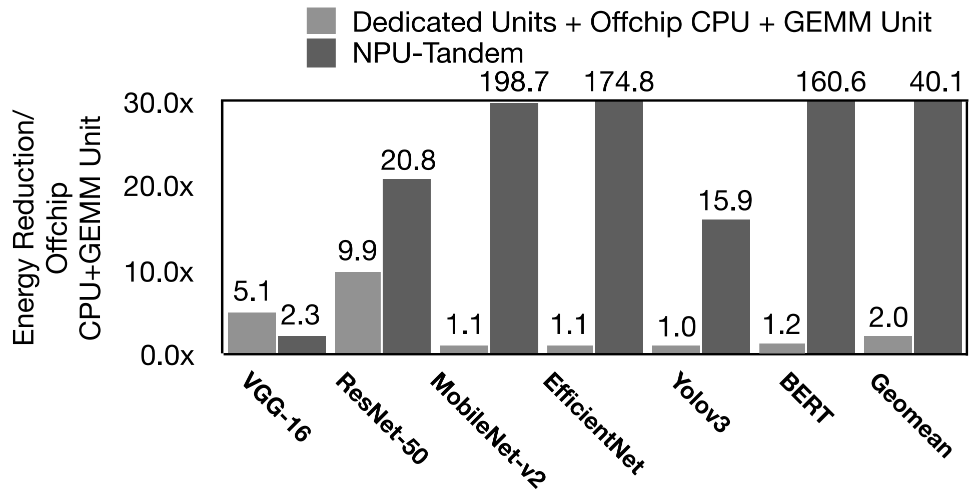


Figure 4.14. Energy comparison to offchip CPU fallback and dedicated units.

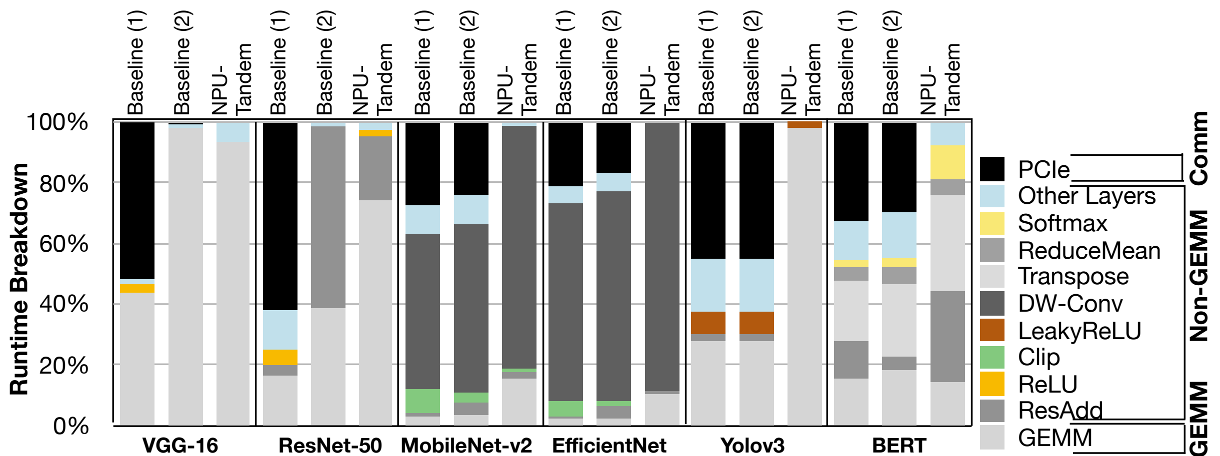


Figure 4.15. Runtime breakdown in NPU-Tandem compared to baseline (1) (Offchip CPU + GEMM Unit) and baseline (2) (Dedicated Units + Offchip CPU + GEMM Unit).

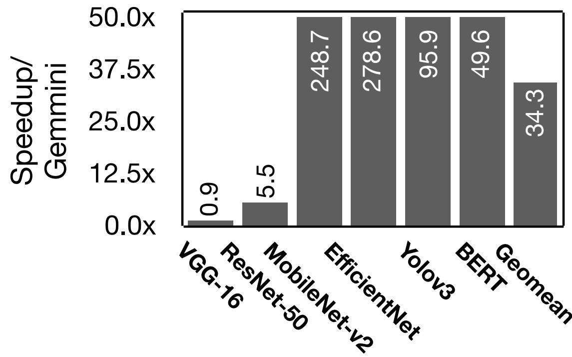


Figure 4.16. Comparison with Gemmini [95].

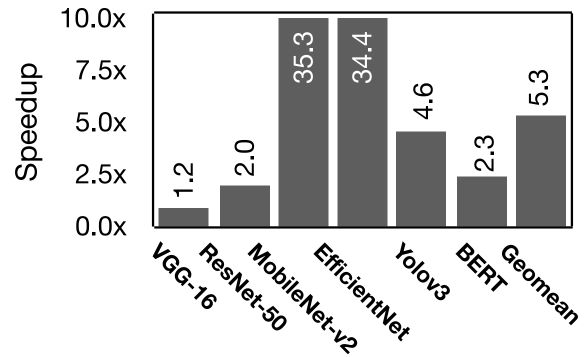


Figure 4.17. Speedup / multi-core RISC-V.

denote the rest as "Other Layers". This analysis sheds light on following insights:

The runtime effects of the non-GEMM operations grow in dominance and they often take the majority of the runtime. As shown in Fig. 4.15, across various DNNs, non-GEMM layers take on average 82% and 68% of the total runtime in baseline (1) and baseline (2), respectively. These results demystify the need for careful execution management of the non-GEMM layers. As Figure 4.15 shows, GEMM layers only become a more significant runtime component, when the highly specialized Tandem Processor is used.

The interweaving of non-GEMM and GEMM operations causes non-trivial overheads. As Figure 4.15 shows, when GEMM and unsupported non-GEMM layers in the baselines are interwoven, the context switching between the GEMM unit and CPU incurs non-trivial overheads due to PCIe communication (See ResNet-50 and VGG-16 for baseline (1) and Yolov3 for baseline (1), (2).).

Non-GEMM layers are in fact very diverse in terms of execution runtime. The more pronounced example here is depth-wise convolution, which is used in MobileNet-v2 and EfficientNet. This layer takes the majority of runtime in baseline (1) and baseline (2) that require to use CPU for its execution. Tandem Processor's specialized memory hierarchy and data access mechanism reduce the cost of depth-wise convolution significantly and as such provides significant performance/energy benefits as shown in Figure 4.13 and Figure 4.14.

Effective execution overlap enabled by tight integration of Tandem Processor and GEMM

unit significantly reduces the runtime. As shown in Figure 4.15, Residual Add, ReLU, and LeakyReLU operations are effectively overlapped with GEMM layers and as such their runtime overheads are reduced to less than 23% in ResNet-50 and 3% in Yolov3 for NPU-Tandem.

Comparison to Gemmini [95]. Figure 4.16 compares the performance of NPU-Tandem with open-source RISC-V integrated Gemmini DNN accelerator [95]. On average, NPU-Tandem provides 34.3× performance improvement over Gemmini. The improvements offered by Tandem Processor are more pronounced for MobileNet-v2 and EfficientNet, since Gemmini needs to transform depth-wise convolution operations to a series of small GEMM operations. This transformation not only requires a time-consuming im2col operation, but also results in low resource utilization on the systolic array. On the other hand, Tandem Processor specialized microarchitecture executes these operations natively and more efficiently without any need for im2col. Also, effective coordination between Tandem Processor and GEMM Unit allows overlapping convolution with depth-wise convolutions which improves the utilization. The improvements for relatively older DNNs (VGG-16 and ResNet-50) are more modest and even for VGG-16, NPU-Tandem slightly underperforms Gemmini due to the support by specialized units.

Figure 4.17 shows the performance improvements over an extended version of Gemmini that integrates the same number of RISC-V cores as the number of SIMD lanes in Tandem Processor. On average, NPU-Tandem provides 5.3× (with maximum of 35.3× for MobileNet-v2 and minimum of 0.9× for VGG-16) speedup compared to this baseline. The benefits due to using multiple cores are more pronounced for BERT and Yolov3 with large number of non-GEMM operations that are not supported by the dedicated units in Gemmini.

Comparison to general-purpose vector integration. We also compare NPU-Tandem to a hypothetical design that integrates Intel i9-9980XE CPU with AVX support with a GEMM unit. Note that this integration is in fact impractical due to the high power consumption of this CPU (165 TDP). As Figure 4.18 and Figure 4.19 show, NPU-Tandem still outperforms this design point by 2.3× in terms of performance and provides 37.5× energy reduction, on average across the

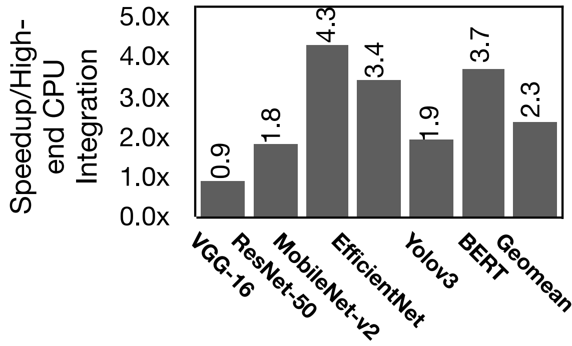


Figure 4.18. Speedup / Intel CPU integration.

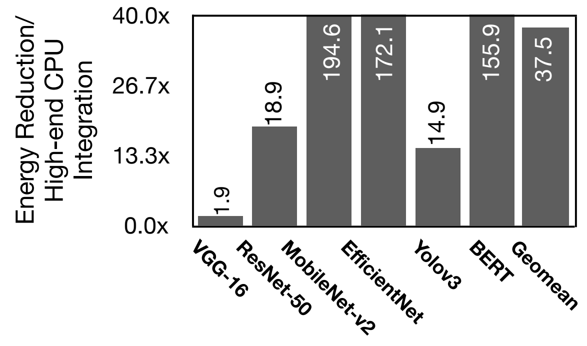


Figure 4.19. Energy / Intel CPU integration.

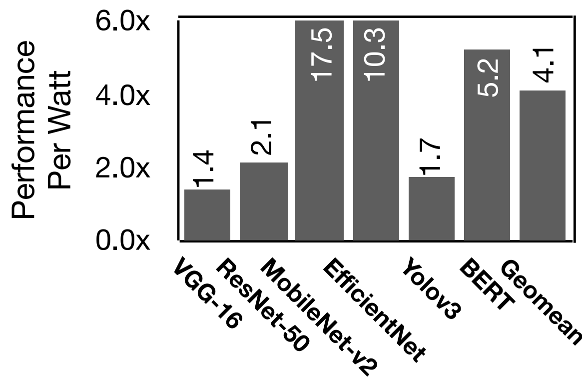


Figure 4.20. Comparison to Jetson Xavier.

Figure 4.21. Fraction of non-GEMM operations to total runtime across various sizes of Tandem Processor.

benchmarks. As the results suggest, although this design point uses an on-chip high-performance general purpose vector processor for handling non-GEMM layers, still the overheads of execution semantics and memory hierarchy in CPU are pronounced, specifically as DNNs evolve over time. To shed light on that, consider VGG-16 and ResNet-50 in Figure 4.18, where Tandem Processor provides 1.8× speedup for ResNet-50 but results in 10% performance drop for VGG-16 compared to high-end CPU. This difference in improvements is mainly due to the augmentation of memory-intensive Residual Add operators in ResNet-50 architecture as opposed to VGG-16, mitigated by customizing the memory hierarchy in Tandem Processor.

Comparison to Jetson Xavier GPU. Figure 4.20 shows the performance-per-Watt benefits over Jetson Xavier NX GPU. On average, NPU-Tandem provides 4.1× improvements while using 12× less number of resources. The trends in the results remain almost similar to the previous

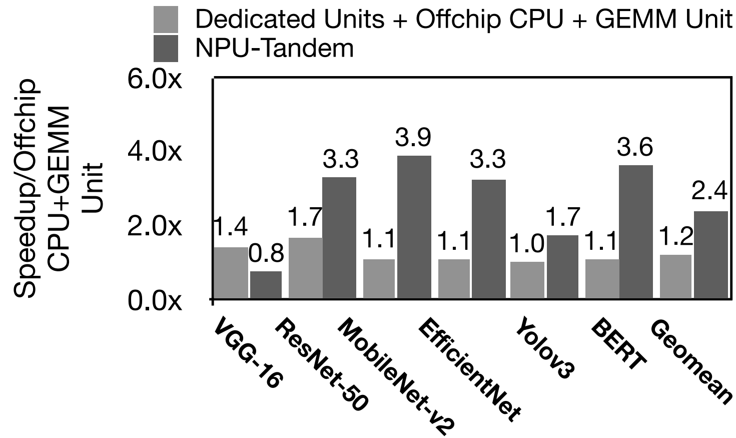


Figure 4.22. Performance comparison to offchip CPU fallback and dedicated units for batch-size=32.

analyses with MobileNet-v2 exhibiting the maximum benefits and VGG-16 seeing the minimum.

Sensitivity to NPU sizes. To analyze the impact of the NPU dimensions to end-to-end runtime breakdown across GEMM and non-GEMM operations, Figure 4.21 sweeps the number of SIMD lanes in Tandem Processor from 8 to 128, while also changing the GEMM Unit size from 8×8 to 128×128 accordingly. Results show that increasing the number of SIMD lanes leads to higher runtime fraction of non-GEMM operations. This is intuitive, since the resources of the GEMM Unit are increased quadraticly as opposed to linear increase in Tandem Processor. These results in fact emphasize the importance of optimizing the non-GEMM operation execution specifically for larger designs. Second, corroborating the analyses in Figure 4.15, the runtime impacts of non-GEMM operations vary significantly across DNN models and it manifests itself even with respect to the NPU size. As shown in Figure 4.21, for ResNet-50 and BERT, the bottleneck of total runtime shifts from GEMM operations to non-GEMM ones by increasing the size of the NPU. On the other hand, for DNNs such as MobileNetv2 and EfficientNet the bottleneck is often non-GEMM operations, while for VGG-16 or Yolov3, GEMM operations take up the majority of the runtime.

Impact of batch size. Figure 4.22 compares the performance of Tandem Processor compared to offchip CPU fallback and dedicated units baselines when batch-size 32 is used. Batching provides increase in the utilization of the baselines by pipelining the GEMM execution, PCIe

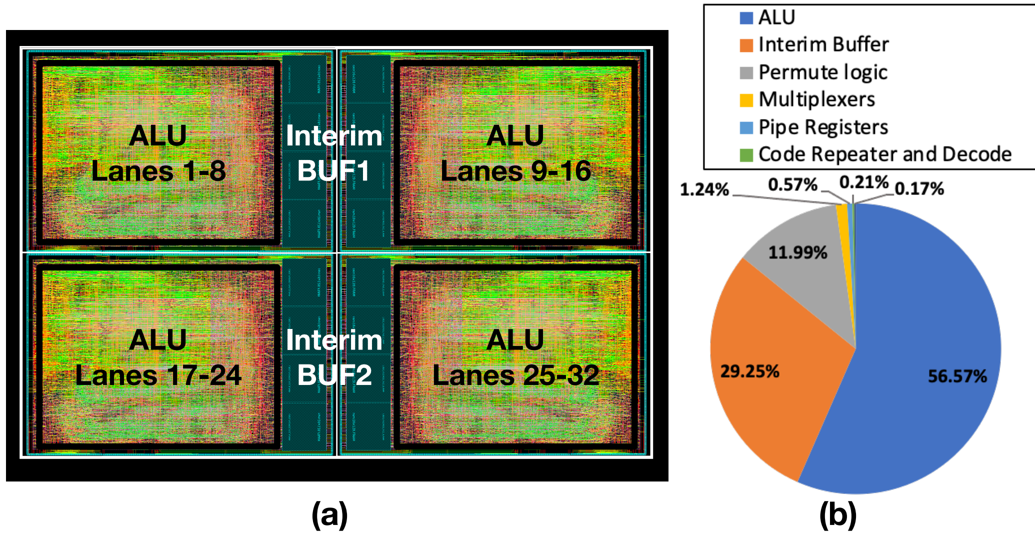


Figure 4.23. (a) Tandem Processor layout and (b) area breakdown.

communication, and CPU execution across images/sequences. As the results show, even when a larger batch size is used, on average Tandem Processor outperforms the offchip CPU fallback and use of dedicated units by 1.2× and 2.4×, respectively.

Tandem Processor layout. Fig. 6.12(a) shows the layout of Tandem Processor in 65 nm, which occupies 1.02 mm^2. Fig. 6.12(b) shows the post-layout area breakdown across major hardware components of Tandem Processor. ALU logic takes the largest portion of area (56.6%), Interim BUF 1 & 2 is the second (29.2%) and the permute logic is the third (12.0%). The rest of the area is mainly for muxing logic, pipeline registers, Code Repeater and decode logic.

4.7 Related Work

There have been a plethora of research works that focus on the acceleration of DNN execution by leveraging various algorithmic and architectural properties. While these works have pushed the frontier in neural acceleration, they largely focus on optimizing the efficient execution of GEMM-based layers through flow of data architecture optimizations [58, 55, 221, 92, 167, 128, 112, 90, 94, 111, 216, 145, 282, 214, 299], data/bit sparsity-aware computation [295, 106, 29, 184, 147, 27, 57, 191, 28, 74, 219, 103, 261, 157, 281], bit-flexible execution [129,

222, 28, 220, 74, 219, 206], near/in-memory acceleration [135, 80, 119, 92, 283] and various other techniques [291, 185, 169, 293, 276, 232, 158, 79, 39, 229]. To support the execution of non-GEMM layers, these prior works have generally taken four approaches: (1) falling back to an offchip CPU [291, 55, 92, 29, 28, 185, 169, 111, 74, 57, 94, 219, 293, 191, 276, 232], (2) employing dedicated hardware units [58, 106, 215, 59, 158, 221, 129, 184, 79, 147, 15, 80, 90, 222, 167, 112, 178, 27, 39, 229, 31, 216, 145, 97], (3) leveraging an on-chip RISC-V microprocessor [95, 239], and (4) using on-chip general-purpose vector units [182, 128, 127, 247]. Section 4.2.3 covers these prior approaches in details and we provide quantitative comparisons in Section 5.6.2. In contrast, this paper exclusively focuses on the impending need for supporting the variegated and ever-expanding non-GEMM operations in DNNs.

4.8 Conclusion

Neural accelerators started with mainly focusing on the performance of GEMM operations, as they formed the main body of earlier neural models. However, the scale is shifting towards novel and ever-growing non-GEMM operations as DNNs evlove and reach to new domains. To address the timely need, this paper proposes Tandem Processor that brings forth a novel architecture and comes with a compiler and an innovative programmable ISA that enables adapting to the volatile landscape of machine learning.

4.9 Acknowledgement

Chapter 4 is a partial reprint of the material as it appears in: S. Ghodrati, S. Kinzer, H. Xu, R. Mahapatra, Y. Kim, B. Ahn, D. Wang, L. Karthikeyan, A. Yazdanbakhsh, J. Park, N. Kim, and H. Esmaeilzadeh, “Tandem Processor: Grappling with Emerging Operators in Neural Networks.” The dissertation author was the primary investigator and author of this paper.

Chapter 5

Cost-Effective Accelerator Utilization via Spatial Multi-Tenancy

5.1 Introduction

The end of Dennard scaling [76] and diminishing benefits from transistor scaling [107, 258, 84] has propelled an era of Domain-Specific Architectures [113]. As such, accelerators are put in the spotlight to enable performance improvements necessary for emerging workloads [105]. Although, most recently, accelerators have made their way into consumer electronics, edge devices, and cell-phones (e.g., Edge TPU [9], NVIDIA Jetson [16], and Apple Bionic Engine [6]), their limited computational capacity still necessitates offloading most of the inference tasks to the cloud. In fact, INFerence-as-a-Service (INFaaS) [202], has become the backbone of the deployed applications in Voice Assistants [11, 7], Smart Speakers [1], and enterprise applications [13, 2, 5], etc. Cloud-backed inference currently dominates the market [12, 3, 4, 8] and is enabled by various forms of custom accelerators, such as Google's TPU [128], NVIDIA T4 [17], Microsoft Brainwave [90], and Facebook's DeepRecSys [102].

As the demand for INFaaS scales, one solution could be continuously increasing the number of accelerators in the cloud. Although intuitive, this approach is neither cost-effective nor scalable with the ever-increasing demand for DNN services. On the other hand, multi-tenancy, where a single node is shared across multiple requests, has been a primary enabler for the success of cloud-computing in current scale. Without multi-tenancy, it is hard to even fathom the progress

and future of datacenters and cloud-based computing. In fact, the broader research community invested more than a decade of efforts to develop solutions across the computing stack to bring forth seamless and scalable multi-tenant cloud execution models [301, 302, 272, 47, 273, 132, 72, 264, 242, 259, 70, 50, 49, 245, 186, 187, 213, 268, 249, 126, 248, 123, 14]. Nonetheless, multi-tenancy has not been a primary factor in the design of DNN accelerators because of the arms race to design the fastest accelerator, the utmost recency of accelerator adoption in datacenters, and challenges associated with multi-tenancy in accelerators. The datacenter accelerator designs revealed–for instance in Google's TPU [128] or Microsoft Brainwave [90]–tend to show results focused on running a single neural network model as fast as possible. Even the MLPerf benchmark suite [197] keeps this single-model focus for both training and inference. But experience in cloud accelerator systems shows that keeping multiple models simultaneously resident on an accelerator has deployment benefits. Beyond just multiple customers sharing an accelerator, there is demand for multi-tenancy inside of a single application. For example, speech recognition and voice synthesis systems tend to require multiple models in deployment and can significantly benefit from multi-tenancy and co-location [20]. Yet, only this year PREMA [62] has explored a scheduling algorithm that time-multiplexes a DNN accelerator across different DNNs through preemption.

This work, on the other hand, sets out to explore this timely, yet unexplored dimension of multi-tenancy in the *architecture design* of DNN accelerators. This work presents Planaria, where the key idea is *dynamically fissioning* the DNN accelerator at runtime to *spatially co-locate* multiple DNN inferences on the same hardware. To that end, the paper makes the following contributions:

1. **Dynamic architecture fission for spatial multi-tenant execution.** This work introduces and explores the dimension of dynamic fission in DNN accelerators. This innovation enables *simultaneous* execution of multiple DNN acceleration threads to be *spatially co-located* on the same hardware substrate. This exclusive runtime reconfigurability in DNN acceleration offers a new degree of freedom in task scheduling to promote utilization and fairness while

meeting the Qulity of Service (QoS) constraints.

2. **Microarchitecture design for dynamic fission.** The work devises a concrete microarchitecture as an instance of dynamic fissionable architectures by delving into the design challenges associated with offering this technology on TPU [128]-like systolic designs. Specifically, we devise omni-directional systolic arrays for DNN acceleration that permits flow of data in all four directions from each elements in the array. This low-cost additional flexibility expands the fission possibilities leading to significant energy reduction and performance gains. To coordinate fission with appropriate on-chip and off-chip data transfer, we arrange these omni-directional systolic arrays in on-chip pods that also comprise specialized interconnection and shared storage for each pod.

3. **Task scheduling for spatial multi-tenant execution.** To leverage architecture-level fission, we define a task scheduling algorithm that breaks up the accelerator with respect to the current server load, DNN topology, and task priorities, all while considering the latency bounds of the tasks. As the following results indicate, this scheduling algorithm can harness fission capability to simultaneously co-locate DNNs to significantly improve utilization, throughput, QoS, and fairness.

We evaluate Planaria using three INFaaS workload scenarios made up of inference requests to nine diverse DNN benchmarks. Each scenario is evaluated under three different Quality of Service (QoS) requirements. We compare the proposed design to PREMA [62], a recent effort that offers multi-tenancy by time-multiplexing the DNN accelerator across multiple tasks. We use the same frequency, the same amount of compute and memory resource for both accelerators. Our results show that Planaria outperforms PREMA in terms of throughput by 7.4×, 7.2×, and 12.2× for soft, medium, and hard QoS constraints, respectively. For these set of constraints, Planaria also offers 45%, 15%, and 16% increase in Service-Level Agreement (SLA) satisfaction rate, respectively. At the same time, Planaria improves fairness by 2.1×, 2.3×, and 1.9×.

Our results suggest that exploring simultaneous spatial co-location through architecture

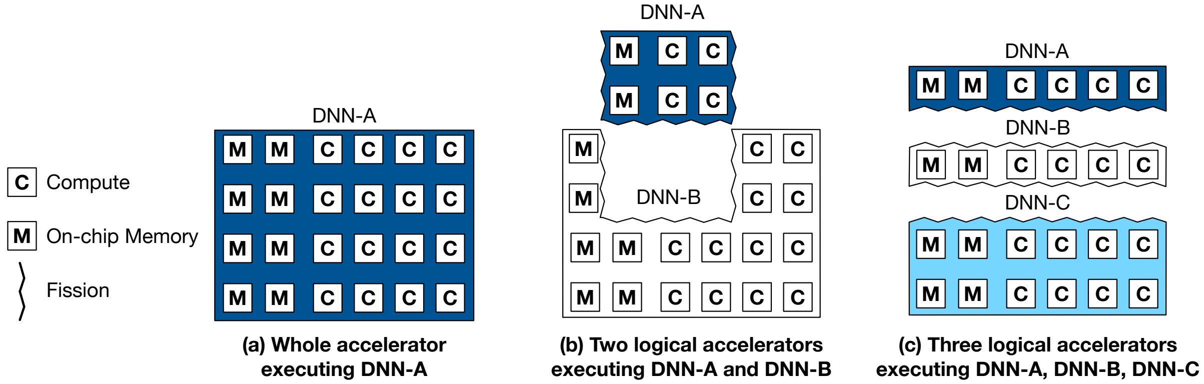


Figure 5.1. Illustration of possible fission schemes of Planaria with their corresponding spatially mapped DNNs.

fission and balanced task scheduling provides significant benefits. To this end, dynamic architecture fission paves the way for spatial multi-tenancy that can offer a unique direction in the era of cloud-scale acceleration of DNNs.

5.2 Dynamic Architecture Fission: Concepts and Overview

The objective is to enable multi-tenant execution of DNNs by spatially co-locating multiple DNN tasks on a single accelerator. To do so, the underlying accelerator needs to dynamically fission at runtime into smaller pieces of *logical full-fledged accelerators* that can execute their pertinent DNN. Figure 5.1 illustrates three possible examples for the proposed accelerator fission and how the accelerator can spatially execute multiple DNN tasks simultaneously. Generally, a DNN accelerator is a collection of on-chip memory banks [M] and compute resources, e.g. Multiply-ACcumulate (MAC) units [C]. Figure 5.1(a) illustrates that if a DNN task with high priority or tight slack to meet the QoS constraint is dispatched to the accelerator, an entire accelerator is dedicated to the task to expedite its completion. In contrast, Figure 5.1(b,c) show multiple DNN tasks being dispatched simultaneously. To process them all, the accelerator can fission into multiple logical accelerators, each of which executes a given task as shown. Importantly, fission needs to take place at both compute and memory level, since each logical engine is a standalone independent DNN accelerator. Moreover, the amount of compute and memory

resources assigned to each logical accelerator ought to be balanced with the computational demand of the dispatched DNNs to maximize the throughput of the accelerator while meeting the QoS constraints. To that end, bringing forth spatial multi-tenant execution requires devising two major components as follows:

Fission microarchitecture. The first component of this work is *a microarchitecture that can fission dynamically into smaller full-fledged accelerators to execute multiple DNNs simultaneously*. Section 5.3 starts from a baseline monolithic DNN accelerator based on systolic array architecture and discusses a set of challenges as well as the design requirements that should be taken into account to fission a monolithic design both at compute and on-chip memory level. Then, Section 5.4 delves into the microarchitectural innards of Planaria, an incarnation of dynamic architecture fission. First, the design of Planaria adds omni-directional data movement in systolic arrays to offer variegated logical fission possibilities. Second, it uses this capability and a unique reorganization of the accelerator, called Fission Pod s, to enable fission in systolic array based DNN accelerators. Fission Pod s are designed to offer a significant degree of fission flexibility, through specialized connectivity, on-chip memory organization, and omni-directional flow of data in its systolic units. This degree of flexibility is necessary to cope with the varying needs of dispatched DNNs that can be best matched by forming heterogeneous logical accelerators as depicted in Figure 5.1.

Task scheduler. As the second component of this work, we devise *a task scheduling algorithm that adaptively schedules and assigns the resources to different tasks.* First, the scheduler identifies minimal amount of resources required to execute the DNN while meeting the QoS constraints imposed. Then, it uses a scoring mechanism that congregates task priority and remaining time to distribute the remaining resources on the accelerator to spatially co-locate tasks. This scoring mechanism leads to higher fairness as it considers multiple criteria and flexibility in the accelerator to co-locate multiple DNNs. Importantly, while the spatial co-location improves fairness, the scheduler effectively utilizes the dynamic fission mechanism and

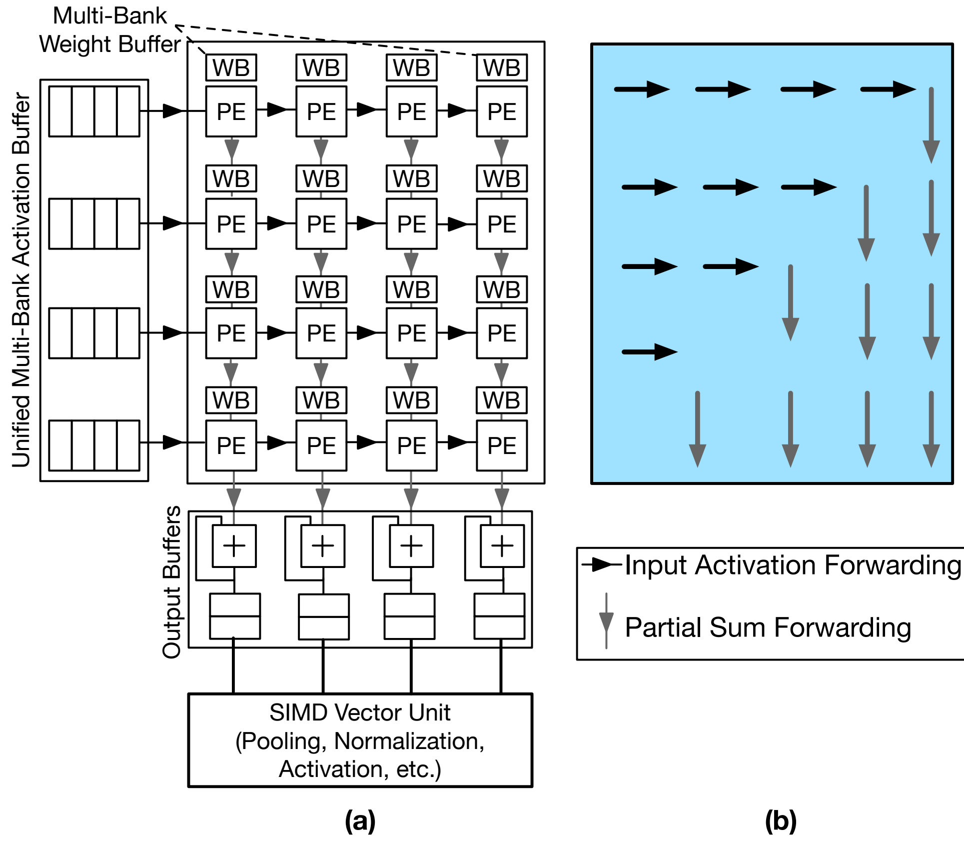


Figure 5.2. **A monolithic systolic array accelerator.**

considers improving the QoS as its primary design principle. In fact, spatial co-location leads to better utilization of the accelerator resources as more than one task can run at the same time. Section 5.5 discusses this scheduling mechanism in detail.

5.3 Architecture Design for Fission: Challenges and Opportunities

This section starts by reviewing a monolithic systolic accelerator, similar to TPU [128]. Then, it provides a series of design requirements to enable spatial multi-tenant execution for DNNs.

Monolithic Systolic Array. Figure 5.2(a) illustrates a monolithic systolic DNN accelerator[1]. The accelerator consists of a 2D array of Processing Elements (PE) to perform matrix multiplications

[1]This section uses a 4×4 systolic array as an example for clarity.

and convolutions, a unified multi-bank Activation Buffer, a 1D array of Output Buffers, and a SIMD Vector Unit to execute the remaining layers such as pooling, activation, batch normalization, and etc. Input activations are stored on-chip in the unified Activation Buffer–generally implemented as a multi-bank scratchpad, where each bank is shared across PEs within a row. Consequently, at each cycle, an input activation is read from an Activation Buffer's Bank and is reused for all the PEs (MAC units) within the row. *At each cycle, each PE forwards the input activation to the PE to its right (horizontal) and the output partial sum to the PE to its bottom (vertical). In short, this is a waterfall-like uni-directional flow of data as illustrated in Figure 5.2(b).* Finally, the outputs are fed to the SIMD Vector Unit for further processing. The remainder of the section elaborates on how to fission all the components comprising this monolithic accelerator.

5.3.1 Fission for Compute and the Need for New Communication Patterns

(1) The need for flexible and cost-effective fission of *compute* resources. Computational characteristics of DNNs such as data reuse and coarse-grained parallelism vary significantly across different networks or even across different layers of a network [147, 94, 57, 146]. The systolic array architectures inherently exploit spatial data reuse for input activations along its rows and partial sums along its columns. *However, a monolithic array design provides only a fixed dimension of this spatial data reuse.* Moreover, as shown for TPU [128], *mapping a convolution or matrix multiplication operation to a big monolithic systolic array can lead to underutilization of compute resources*. As such, some layers naturally perform better if they are tiled to smaller chunks and parallelized across multiple smaller arrays, as that would exploit coarse-grain parallelism and yield better resource utilization.

Figure 5.3 illustrates multiple examples of possible configurations for decomposition of a 4×4 systolic array, where a 2×2 subarray is used as the granularity for fission. Figure 5.3(a) shows a fission where the systolic array is broken down horizontally into two subarrays, while Figure 5.3(b) shows an instance of its vertical fission into two subarrays. Figure 5.3(c) illustrates

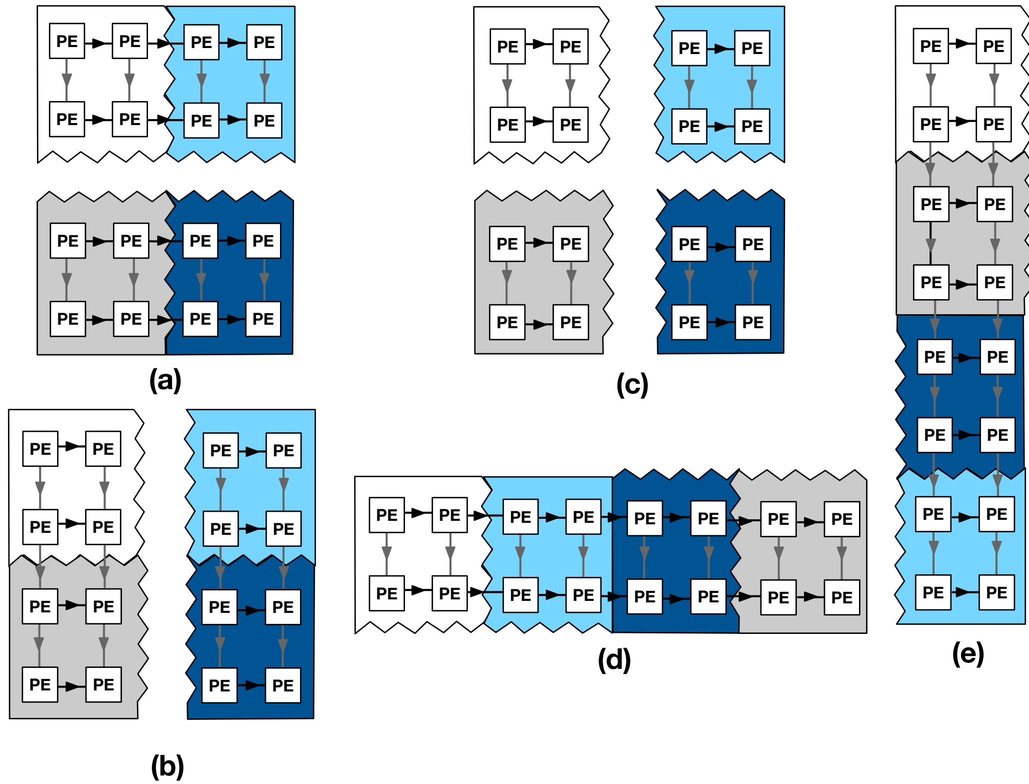


Figure 5.3. Illustration of possible fission scenarios.

another fission in both vertical and horizontal directions, yielding four systolic subarrays. For a layer that requires high coarse grain parallelism, fission in Figure 5.3(c) would be a good match, while Figure 5.3(b) would yield the best performance for layers that enjoy more partial sum reuse as well as the coarse-grain parallelism. In another scenario, if a layer requires high input activation reuse, moderate partial sum reuse, and coarse-grain parallelism, fissioning to Figure 5.3(a) will be the best choice.

Fission granularity. With respect to compute fission, an important design decision is where to break the systolic array. As Figure 5.2 shows, PEs are connected via two uni-directional links: horizontal and vertical. One extreme option is to replace these links to those that can be dynamically switched on and off to fission the systolic array at the granularity of a single PE. However, such a fine granularity of fission will impose significant overheads. Therefore, *we instead replace a subset of the links to determine the granularity such that they can disconnect a*

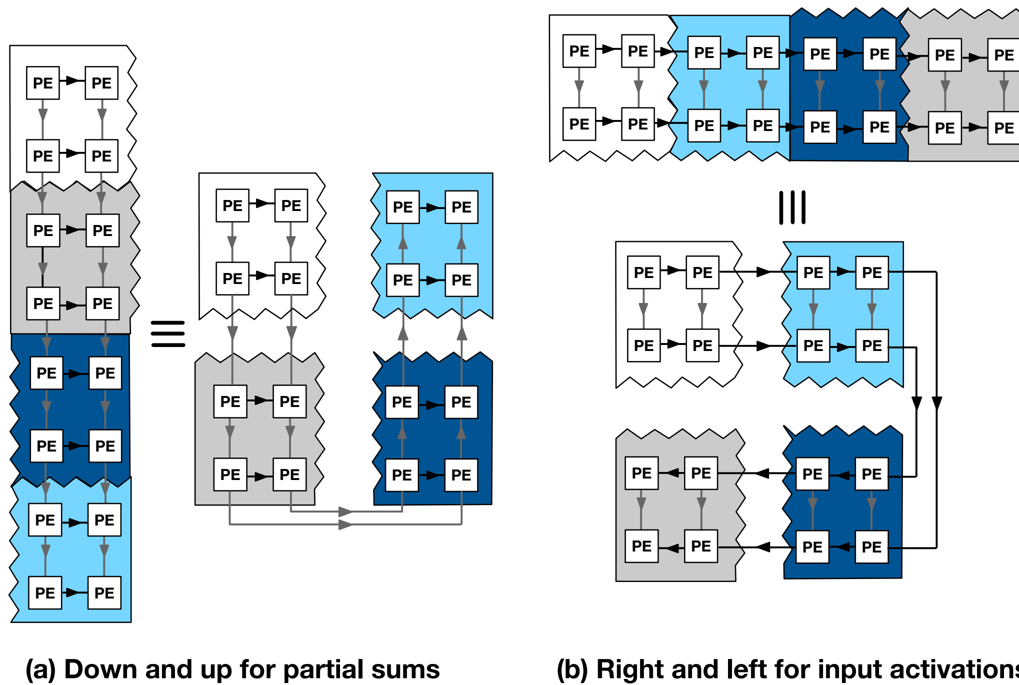


Figure 5.4. Omni-directional systolic execution.

subarray of the PEs instead of a single PE. The design space exploration of the subarray size is discussed in Section 5.6.2.

(2) The need for new and flexible patterns of *communication* for richer fission possibilities. Figure 5.3(d,e) illustrates two more fission scenarios. If a network layer provides significantly higher opportunity in input activation reuse than partial sum reuse, while not requiring high parallelism, a scheme such as Figure 5.3(d) is desirable, while fissioning to Figure 5.3(e) will be a better design for significantly high partial sum reuse. Realizing the last two configurations, however, requires additional design considerations. To forward the input activations along four subarray fragments in Figure 5.3(d) and partial sums in Figure 5.3(e), the data needs to flow in all directions: right and left for input activations and up and down for partial sums. Figure 5.4(a) and Figure 5.4(b) illustrates how the partial sums and input activations need to flow at all directions to realize the desired scenario. To that end, *we propose omni-directional systolic arrays that can forward the input activations and partial sums in all directions as opposed to conventional systolic arrays that always forward the data in just two directions.*

Communication across the fissioned subarrays. In addition to the omni-directional intra-subarray data movement, there is a need for low-cost inter-subarray communication that also facilitates reconfigurability. As such, *we propose a bi-directional ring bus to connect the fissioned systolic subarrays instead of other forms of connectivity, e.g. crossbar, that would impose significant overheads. The bi-directional nature is to extend omni-directional communication along the subarrays. The links of the ring are configurable in that they can be either off to fission two subarrays or on to forward input activations and partial sums.*

(3) Enabling full-fledged logical accelerators through fission for the SIMD Vector Unit . To create stand-alone accelerators from the fissioned units, the SIMD Vector Unit also needs to be broken into smaller segments and coupled with each systolic subarray. Due to the parallel nature of this unit, we divide the original SIMD Vector Unit to smaller segments proportional to the number of systolic subarrays, and designate a segment to each. When systolic subarrays are vertically stacked (e.g., Figure 5.3(b,e)), a subset of these SIMD segments are bypassed.

5.3.2 Fission for the On-Chip Memory and the Need for Reorganizing the Entire Design

Besides the systolic array, the accelerator also requires fissioning the on-chip memory blocks to allocate commensurate storage to the compute units. Memory disaggregation across the chip is crucial for maximizing on-chip resource utilization. That is because, the on-chip buffers bandwidth to the PE subarrays needs to be kept unchanged to supply enough data to keep the PEs busy. Otherwise, fission would diminish utilization instead of improving it, which was a primary objective of this work.

While decomposing Weight Buffer is straightforward due to its coupling within the PEs, fission for the Activation Buffer and Output Buffer is more challenging.

Weight buffer fission. In systolic arrays, each PE harbors a private Weight Buffer that holds a subset of the network parameters. As such, the total Weight Buffer gets broken down naturally during fission as our strategy does not break the PE.

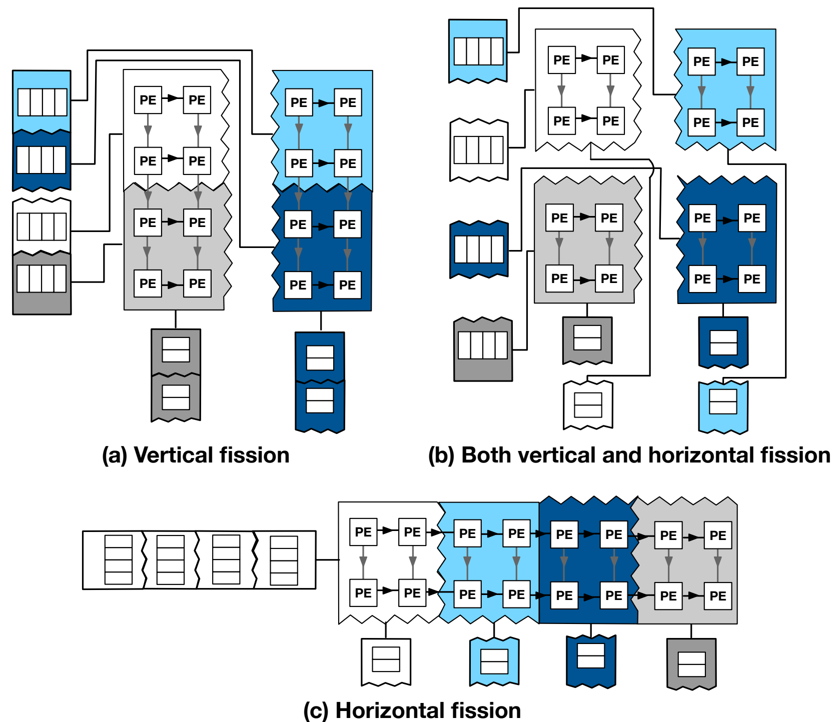


Figure 5.5. On-chip memory fission and connection to subarrays.

Activation and output buffer fission. Figure 5.5 illustrates on-chip memory fission for three of the scenarios shown in Figure 5.3(b,c,d). Each of the scenarios requires different fission scheme for the Activation Buffer and Output Buffer as well as various patterns of connection between the buffers with the systolic subarray, which are not possible in a monolithic design. In the monolithic case, the Activation Buffer is just connected to the leftmost PEs and Output Buffer to the bottom-most PEs. However, as Figure 5.5 depicts, more patterns of connectivity between these buffers and the PEs/subarrays are necessary. To support these variegated patterns, *we reorganize the entire accelerator and devise a microarchitectural block, dubbed Fission Pod , where the Activation Buffer and Output Buffer are co-located in a dedicated memory substrate that is shared amongst a group of connected omni-directional systolic subarrays.* This reorganization and sharing a dedicated memory substrate amongst a *group of subarrays* is crucial to strike a balance between the cost of connectivity in the hardware and achieved utilization of the compute and memory resources.

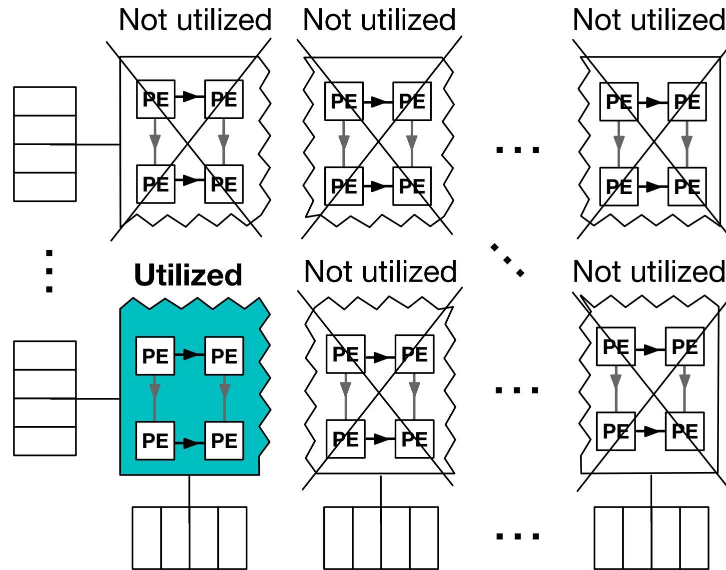


Figure 5.6. Underutilization of the subarrays while they are connected to on-chip memory similar to conventional systolic arrays without reorganization of the design. The teal-colored subarray is the only one that can be utilized.

5.3.3 Fission without Reorganization Defeats the Purpose

Figure 5.6 illustrates a hypothetical case when the systolic array has been partitioned into multiple independent subarrays without properly reorganizing the memory modules. As shown, only the subarray at the left-bottom corner could be utilized, as it would be the only one connected to the Activation Buffer and Output Buffer banks. The other subarrays could not be utilized and would remain idle as illustrated in Figure 5.6. This underutilization would be a common case if fission happens at granularities other than a single subarray.

Another extreme is illustrated in Figure 5.7 illustrates where an alternative hypothetical design point connects all the Activation Buffer and Output Buffer banks to all the subarrays. As depicted, this design would require *two high-radix* $n \times n$ *crossbars*, where n is the number of subarrays. This significantly costly solution is necessary to provide the connectivity patterns discussed in Figure 5.5 and avoid underutilization of the subarrays. This design point is also not acceptable due to the high-radix crossbars, and can seriously curtail scaling up the compute resources.

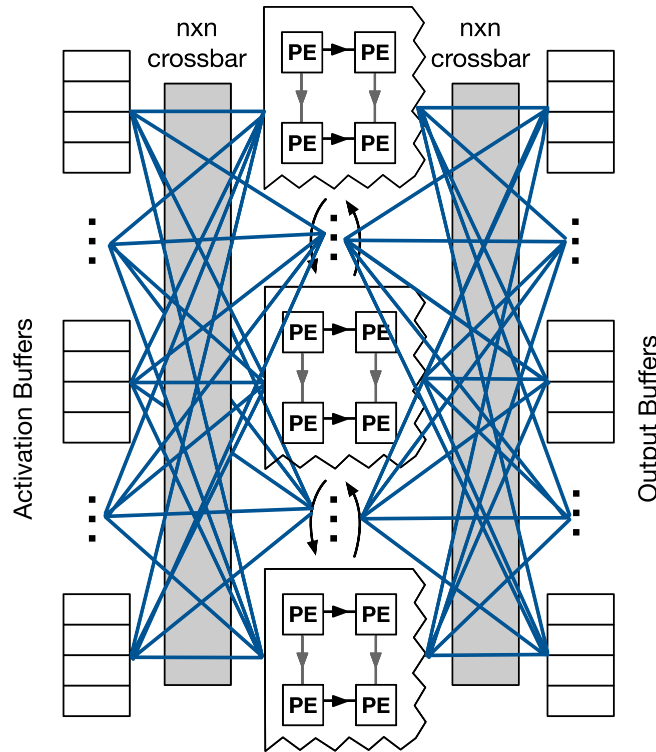


Figure 5.7. On-chip memory to subarrays connectivity through high-radix crossbars in an alternative hypothetical design point.

Our Fission Pod which reorganizes the subarrays and on-chip memory amortizes this significant overhead, while providing the connectivity patterns required to achieve high utilization of the computer resources as discussed in the next section.

5.4 Microarchitecture for Fission

This section delves into the microarchitecture design of dynamic architecture fission for spatial multi-tenant execution.

5.4.1 Omni-Directional Systolic Array Design

Our novel insight is that, for fission, there is a need for omni-directional pattern of communication in the systolic array to enable richer fission and rearrangement possibilities. An opportunity exists to support this pattern through addition of a low-cost logic to each PE. Figure 5.8 illustrates how a set of of additional multiplexers around a PE can enable this omni-

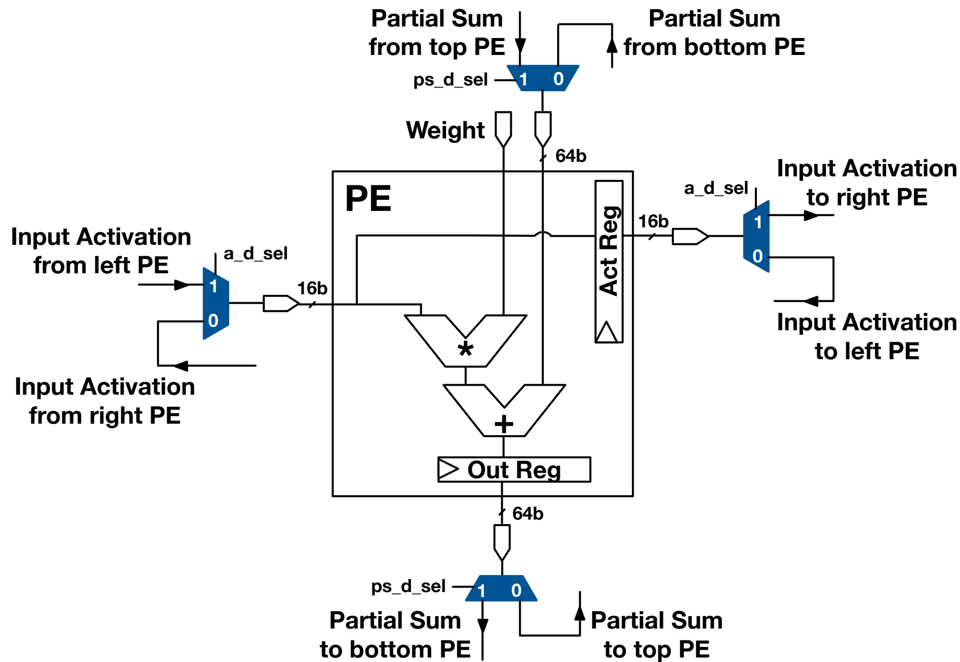


Figure 5.8. Switching network for omni-directional systolic array.

directional movement. In addition to the normal flow of data (right and down), these multiplexers enable each PE to send input activations to its left and partial sums to the PEs at its top. As highlighted in dark blue, a multiplexer at the left of the PE selects its input from either the activation coming from the right or the left. A de-multiplexer at the right selects which of the left or the right PE should receive the activation. The multiplexer and the de-multiplexer are coupled and are controlled by the same single bit, setting the direction of input activations along the array. Similarly, another pair of multiplexer/de-multiplexer on the north and south of the PE in the Figure 5.8 control the flow of partial sums. To enable fission and omni-directional inter-subarray data movements, PEs at the boundaries of systolic subarrays are also connected through these multiplexer/de-multiplexer pairs to the corresponding PEs in the adjacent subarrays.

Effect on clock frequency. Synthesis results show that this extra logic does not reside on the critical path that determines the clock cycle of the systolic subarray. In fact, the critical path is from the Weight Buffer to the Output Register, as access to the on-chip buffer dominates the execution time.

5.4.2 Reorganizing the Accelerator Microarchitecture through Fission Pod Design

The key objectives in designing the microarchitecture for dynamic fission are:

1. Creating multiple stand-alone and full-fledged logical accelerators to enable spatial co-location.
2. Enriching the fission possibilities as much as possible to serve various computational needs of co-located DNNs.
3. Maximizing the PE subarray utilization.
4. Maximizing the on-chip buffers utilization and their bandwidths to subarrays.

While meeting these design objectives, the following design constraints need to be considered:

1. Imposing minimal power/area overhead to the hardware
2. Maintaining the baseline clock frequency

With these design objectives and constraints in mind, we propose a microarchitectural unit, called Fission Pod , which *interweaves the on-chip memory with the systolic subarrays and provides balanced cooperation of these components*. Figure 5.9 illustrates the design. As shown, at the center of this unit, an on-chip memory substrate, called Pod Memory , is placed and connected to a *group* of systolic subarrays. Following discusses the cooperation and communication of the subarrays and the Pod Memory .

Memory-compute interweaving in Fission Pod . A conventional systolic array harbors a unified multi-bank Activation Buffer and a unified multi-bank Output Buffer on their left and bottom, respectively (see Figure 5.2). When a systolic array is broken into four subarrays as depicted in Figure 5.9, the aforementioned buffers are moved to Pod Memory and are broken down into

four corresponding independent multi-bank buffers. These four buffers are connected to the four systolic subarrays via two 4×4 crossbars to maximize flexibility and fission possibilities that require various patterns of connectivity between on-chip buffer and a group of (size four in this work) systolic subarrays, while also maximizing the PE subarray and on-chip buffer utilization. One crossbar is for reading from Activation Buffer s and the other for writing to Output Buffer s of the Pod Memory . This design point is in contrast with the design shown in Figure 5.7, where all the subarrays are connected *globally* to all on-chip buffers through high-radix crossbars, leading to significant power/area overheads, as in here lower-radix crossbars are sufficient due to reorganizing a group of subbarrays and on-chip buffers in a Fission Pod (first design constraint).

Intra Fission Pod data communication. The systolic subarrays are also connected to one another via two sets of bi-directional ring buses. One bus is to pass activations between omni-directional subarrays (❷) and the other is to forward the subarray partial sums (❸). These buses enable realizing different fission possibilities while leveraging the omni-directional feature of proposed subarrays. For instance, to realize the fission scheme in Figure 5.3(d), where the subarrays reconstruct a fat and short array, the activation ring bus will chain the subarrays. The SystolicSubarray-0 in Figure 5.9 sends the activations to SystolicSubarray-1, and so on and so forth. Since for fission scheme in Figure 5.3(d), there is no need for partial sum forwarding, the partial sum ring bus will be switched off.

Clock frequency consideration. The two ring buses are pipelined with 12 registers to alleviate any potential critical paths due to the connectivity between the subarrays. Pipelining is feasible due to natural behavior of systolic arrays that pump wavefronts of data continuously. As such, the added connectivity and switching mechanisms does not result in altering the baseline frequency (second design constraint).

5.4.3 Planaria Overall Architecture

Figure 5.10 illustrates the overall architecture of our proposed accelerator, Planaria. As shown, the original monolithic systolic array has been broken down into 16 omni-directional

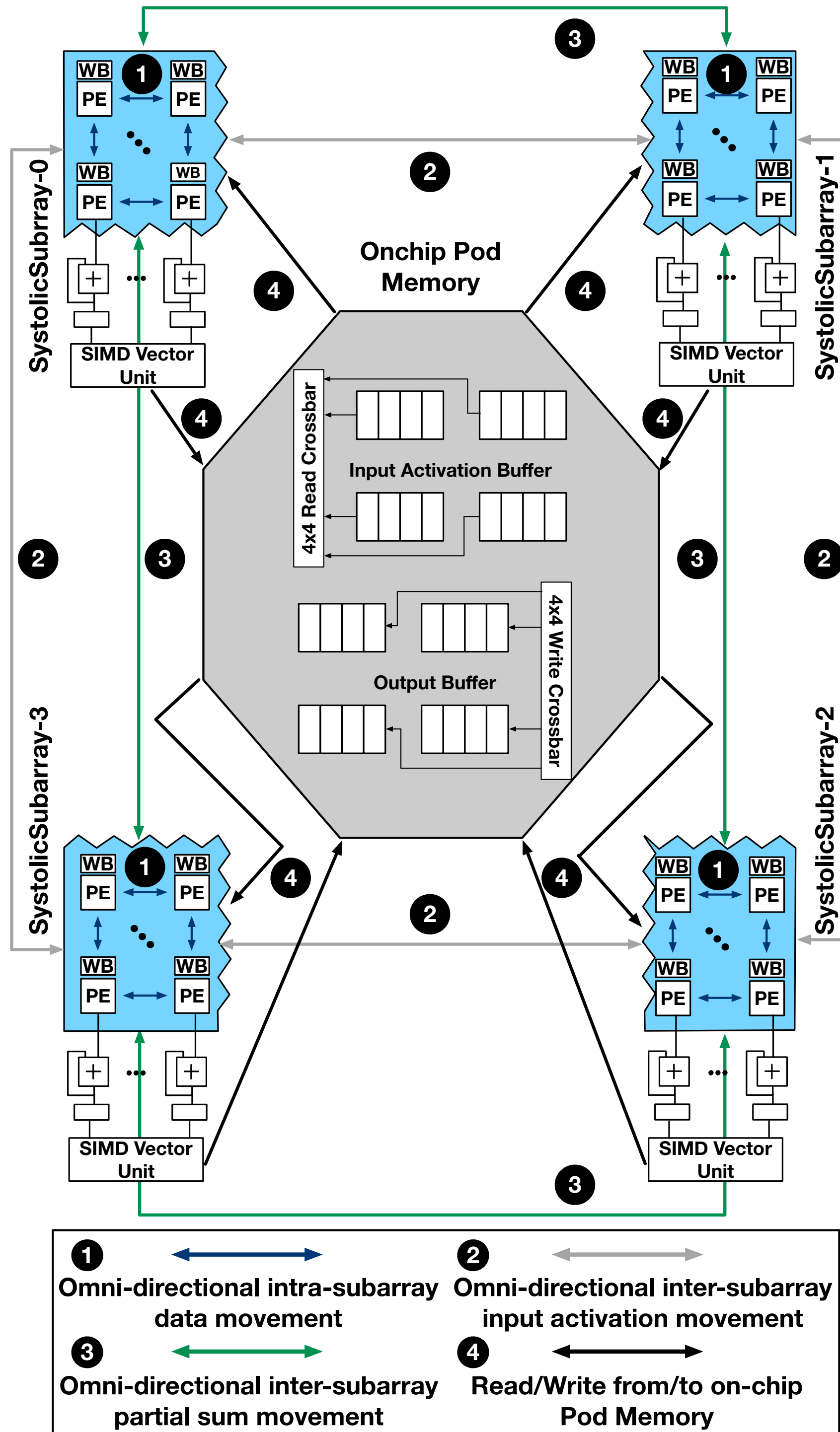


Figure 5.9. Fission Pod .

systolic subarrays, where a group of four subarrays form one Fission Pod that contains a Pod Memory . All these 16 subarrays are connected globally along the accelerator chip via the aforementioned bi-directional ring busses for input activations and partial sums data movement. Hence, in one extreme, all these ring busses can be switched on to construct the biggest logical accelerator, running only one DNN on the entire accelerator. Alternatively, in another extreme, all of the ring buses can be switched off to provide 16 standalone logical accelerators, spatially co-locating 16 different DNNs simultaneously for multi-tenant execution. Overall, this architecture supports 65 fission scenarios that can simultaneously co-locate various number of DNNs from 1 to 16. Each of the four Fission Pod s is connected to one off-chip memory channel. The bus that brings the data from off-chip memory channel simply goes around the subarrays and can fill their weight buffers. This bus is also connected to the Pod Memory to load/store intermediate activations/output to/from the off-chip memory channel. This bus is pipelined and is no different than the bus that feeds the off-chip data to a conventional systolic array. To avoid clutter, Figure 5.10 does not illustrate the off-chip memory buses. The Fission Pod s are connected to their neighbors through a direct link that can foster data reuse to reduce costly off-chip accesses. If data is present in one of the pods, it can be sent to another at most with two hops.

Planaria can fission up to 16 logical accelerators and therefore, it can simultaneously co-locate 16 different DNNs. However, depending on the combination of the co-located DNNs, 65 total fission scenarios are possible. A logical accelerator, which represents one of these 65 possibilities, can encompass multiple physical Fission Pod s. A logical accelerator can either work as a logical monolithic systolic array or further fission if a DNN layer benefits from coarse-grain parallelism. Planaria's interconnections and bus are designed such that, a logical accelerator can take a portion of a Fission Pod and another logical accelerator takes the rest. In Figure 5.10, one logical accelerator that accelerates DNN_A can comprise the subarrays in Fission Pod-0 with two subarrays from Fission Pod-3 (Fission Pod-3.SystolicSubarray-0 and Fission Pod-3.SystolicSubarray-1). The remaining two subarrays from Fission Pod-3 can form another logical accelerator to accelerate DNN_B.

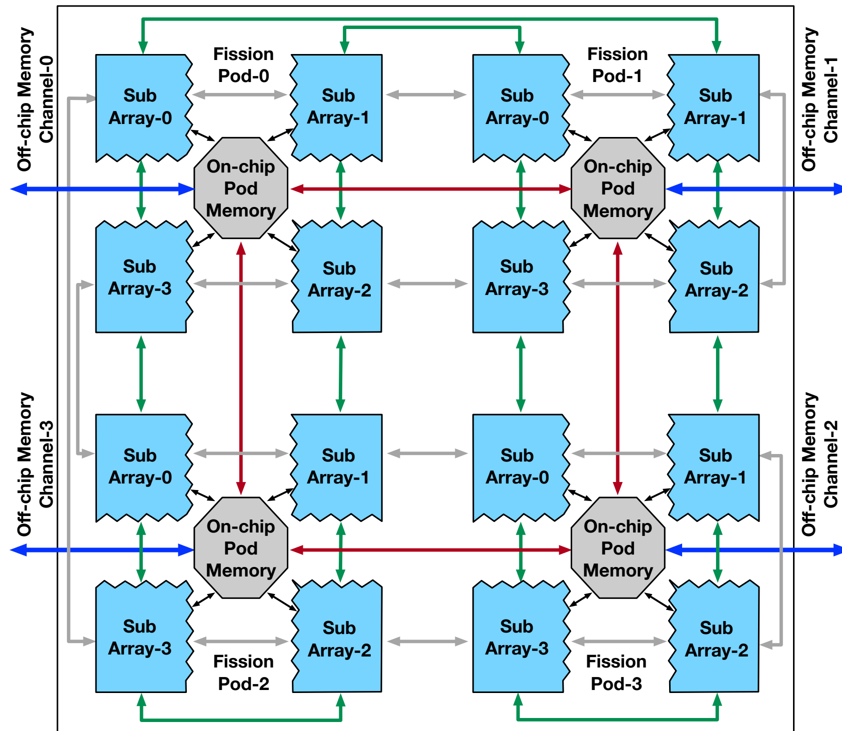


Figure 5.10. Overall architecture of Planaria.

Dynamic reconfiguration for fission and multi-tenant execution. Conventional systolic arrays operate in tile granularity. That is, they fetch a tile of weights and activations and produce a tile of intermediate activations or outputs. Planaria does not deviate from this convention. Consider a scenario where three DNNs are simultaneously co-located with some fission scheme on Planaria, and fourth DNN is now dispatched to be accelerated. In this case, Planaria allows the old three co-located DNNs finish computing the tile that they are processing. In the meantime, the scheduler decides the new allocation of the subarrays considering the newly dispatched DNN. At the same time, Planaria loads this new fission configuration as a set of bits that decides the direction of the subarrays and the off/on connectivity state of the buses. Each Planaria subarray requires two 6-bit registers, one retaining the current configuration state and the other pre-holding the next state. Six bits is sufficient for reconfiguration of each subarray and its directions/buses. Two bits determines the direction of input activation and partial sums. Each subarray can potentially connect to four other subarrays, which can be in the neighboring Fission

Pod s, determined by four bits. The direction of connectivity can be deduced from the direction of the subarray. Another eight bits determine the connectivity of the Pod Memory buffers to the subarrays in the same Fission Pod . Similar to conventional systolic design, each subarray is equipped with an instruction buffer and a Program Counter, indicating the current macro instruction. While the subarray is draining the instructions for the old DNNs, Planaria fetches the next instructions associated with the new configuration. The mechanism is no different than prefetching the instructions for a new tile in conventional systolic arrays. The difference is that, each subarray has a designated PC and a designated 4 KB instruction buffer.

Compilation for Planaria. For INFaaS, since each DNN will serve unbounded set of inference requests, it is intuitive to precompile the DNN and run the precompiled binary again and again. Figure 5.11(a) illustrates the workflow of Planaria compiler. As the DNN may be allocated different number of subarrays (from 1 to 16) during its execution on Planaria, the compiler generates a total of 16 binaries and 16 configuration tables per DNN to cover all the possibilities. The compiler is aware of all the possible architecture fission configurations at compile time. As such, it iterates over all possible configurations to identify the optimal fission configuration as well as the corresponding tiling sizes. Importantly, as different layers vary in parallelism and data reuse, so does the optimal fission configuration for the layer. Therefore, for each layer, *in an offline manner*, the configuration table stores the optimal fission configuration, the number of tiles, and the estimated number of cycles per tile. Such estimation is viable because dataflow graphs of DNNs are fixed, there is neither control flow speculation, nor hardware manged cache to cause significant variation in the latency. At runtime, the proposed scheduler which runs on the host CPU uses this software table of estimates to perform QoS-aware scheduling and resource allocation.

5.5 Spatial Task Scheduling

Dynamic architecture fission adds a new dimension in task scheduling, and provides opportunity to break the DNN accelerator into multiple logical accelerators to not only co-locate multiple tasks, but also to provide logical accelerators tailored for the needs of DNN tasks to promote utilization and consequently throughput, SLA satisfaction rate, and fairness. To that end, the scheduler needs to take into account the following requirements:

1. The scheduler ideally needs to be aware of the optimal fission configurations for DNN tasks to leverage dynamic fission and co-location.

2. The scheduler needs to be *QoS-aware* and leverage the available slack time offered by QoS constraint of each task to maximize the co-location and utilization while adhering to the SLA.

3. Task re-allocation requires checkpointing the intermediate results, while making sure that the re-allocation and checkpointing does not overuse on-chip memory or result in significant context switching overheads.

With these requirements, this section delineates the overall flow of our proposed spatial task scheduling (Algorithm 1).

Overall flow. To leverage the dynamic architecture fission, the scheduler is invoked whenever (1) a new inference task is dispatched to the task queue (Ω in Algorithm 1) of the datacenter node or (2) a running inference task finishes. Each scheduling event consists of the following two major stages. Given the DNNs in the queue, the first stage determines the minimum amount of resource (number of subarrays) necessary to meet the QoS requirements for each task. Given that, the second stage determines the allocation of the subarrays based on their availability and priority of the inference requests. This high level flow of the scheduling is shown in function SCHEDULETASKSSPATIALLY.

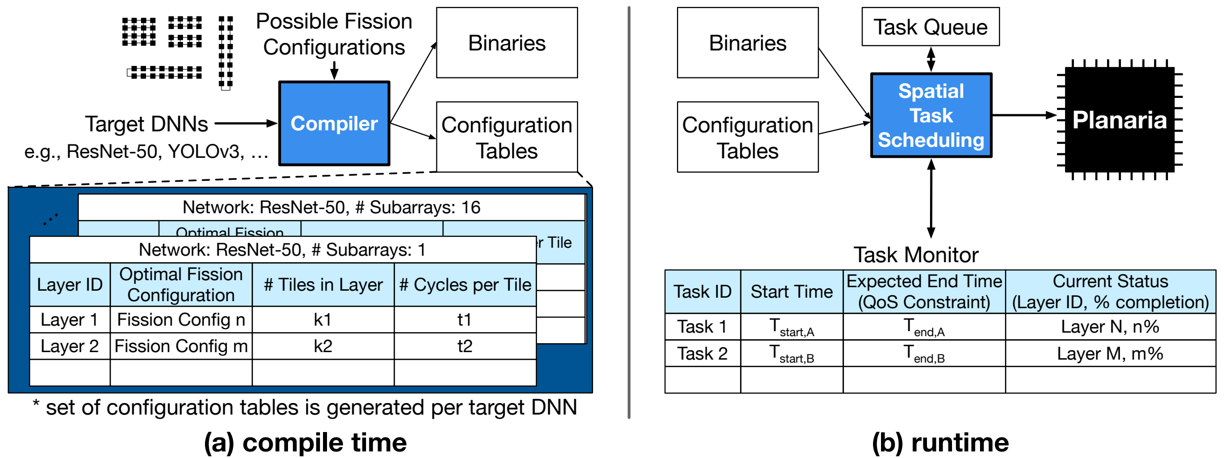


Figure 5.11. Overall workflow.

Estimating minimal resource to meet the QoS requirement. This algorithm exploits the dynamic architecture fission by adaptively assigning resources with regard to the intrinsic slack times provided by each DNN inference task. The algorithm begins by first identifying the minimal resources required to meet the QoS requirement. As illustrated in Figure 5.11(b), spatial task scheduler utilizes a task monitor to keep track of the running tasks. As shown, the configuration tables generated during compilation is used in conjunction with the current status of the task to predict its remaining time. Thus, the PREDICTTIME function reduces to merely looking up the number of remaining tiles with their cycles and performing simple calculation. Then, the scheduler uses the prediction for each configuration to determine the minimum number of subarrays for each task. The ESTIMATERESOURCES function in Algorithm 1 summarizes this stage.

Allocating resources to improve QoS. After identifying minimal resource for each task, the scheduler determines whether all the tasks in the queue can be co-located simultaneously. Depending on whether or not all the tasks can be spatially co-located on Planaria, this stage invokes two different functions, ALLOCATEFITTASKS and ALLOCATEUNFITTASKS, as shown in line 6–10 in Algorithm 1. First, when all the tasks can be spatially co-located, the function ALLOCATEFITTASKS will first assign the minimum number of subarray required to meet the QoS requirements. Then, if there are remaining resources, the scheduler aims to optimally

Algorithm 1. Spatial Scheduling for Planaria

```
 1: function SCHEDULETASKSSPATIALLY(Ω):
 2:     estimates ← {}
 3:     for task in Ω do
 4:         estimates[task] ← ESTIMATERESOURCES (task)
 5:     end for
 6:     if Planaria.fits(estimates) then
 7:         s ← ALLOCATEFITTASKS (Ω, estimates)
 8:     else
 9:         s ← ALLOCATEUNFITTASKS (Ω, estimates)
10:     end if
11:     return s
12: end function
```

```
13: function ESTIMATERESOURCES(task):
14:     candidates ← [ ]
15:     slack = task.constraint - task.executed_time
16:     for num_subarray in range(Planaria.size) do
17:         if PREDICTTIME(task) ≤ slack then
18:             candidates.append(num_subarray)
19:         end if
20:     end for
21:     return min(candidates)
22: end function
```

```
23: function ALLOCATEFITTASKS(Ω, estimates):
24:     allocation ← { }, scores ← { }
25:     for task in Ω do
26:         allocation[task] ← estimates
27:         scores[task] ← task.priority / task.remaining_time
28:     end for
29:     remaining_array ← Planaria.size - ∑estimates
30:     for task in Ω do
31:         fraction ← scores[task] / ∑scores
32:         allocation[task] += fraction×remaining_array
33:     end for
34:     return allocation
35: end function
```

```
36: function ALLOCATEUNFITTASKS(Ω, estimates):
37:     allocation ← { }, scores ← { }
38:     for task in Ω do
39:         slack ← task.constraints - task.executed_time
40:         scores[task] ← task.priority / (slack×estimates[task])
41:     end for
42:     scores.sort(reversed=True)
43:     remaining_array ← Planaria.size
44:     while remaining_array > 0 do
45:         allocation[task] ← estimates[task]
46:         remaining_array -= estimates[task]
47:     end while
48:     return allocation
49: end function
```

distribute these spare resources using a score function that balances priority and the remaining time of each task, as shown in line 27 in Algorithm 1. Consequently, this score function not only fosters throughput but also the fairness among the tasks. Finally, the scheduler allocates the spare resources proportional to the score of each task.

On the other hand, when only subset of the tasks fit on Planaria, the scheduler uses the ALLOCATEUNFITTASKS function to resolve the competition among the tasks. Similar to the approach used to assign the spare resources, the function leverages a score that uses priority, slack, and the minimum required resource of each task, as shown in line 40 in Algorithm 1. This scoring mechanism gives advantages to the tasks with higher priority to improve fairness, and to the ones with less slack time or less resource requirement to maximize QoS satisfaction and throughput. Finally, the scheduler allocates the resources to different tasks in the order of their scores until Planaria becomes fully occupied.

Tile-based scheduling to minimize re-allocation overheads. To prevent the running tasks from stalling, (1) the scheduling happens at tile-granularity and (2) the tasks are preempted only when the resource allocation changes. These two strategies in tandem minimizes the potential preemption delays that may reduce throughput. Moreover, the tile-based scheduling minimizes the memory requirements for preemption as only a single tile of intermediate results needs to be stored off-chip. This in turn obviates the need for additional on-chip storage to support preemption.

5.6 Evaluation

5.6.1 Methodology

Benchmark DNNs. Following the MLPerf [197] methodology, we choose our representative DNN models from domains of image classification [110, 241, 244, 116], object detection [162, 198, 199], and machine translation [274]. We use nine diverse DNNs from these domains to construct a set of DNN tasks with various layer dimensions and types of operations including

Table 5.1. **Workload scenarios and benchmark DNNs from three domains: image classification [110, 241, 244, 116], object detection [162, 198, 199], and machine translation [274].**

Workload	Load Weight	Domain	DNN Model (Release year)
Workload Scenario-A	Heavier	Image Classification	ResNet-50 (2015), GoogLeNet (2014)
		Object Detection	YOLOv3 (2018), SSD-R (2016)
		Machine Translation	GNMT (2016)
Workload Scenario-B	Lighter	Image Classification	EfficientNet-B0 (2019), MobileNet-v1 (2017)
		Object Detection	SSD-M (2017), Tiny YOLO (2017)
Workload Scenario-C	Mixed	Image Classification	ResNet-50, GoogLeNet, EfficientNet-B0, MobileNet-v1
		Object Detection	YOLOv3, SSD-ResNet34, SSD-MobileNet, Tiny YOLO
		Machine Translation	GNMT

recent and state-of-the-art deep neural models such as EfficientNet and YOLOv3.

Multi-tenant workloads. Commensurate with MLPerf, as Table 5.1 shows, we create three INFaaS workload scenarios made up of inference requests to the benchmark DNNs: (a) Workload-A (from requests to ResNet-50 [110], GoogLeNet [241], YOLOv3 [199], SSD-R [162], and GNMT [274]); (b) Workload-B (from requests to EfficientNet-B0 [244], MobileNet [116], SSD-M [162], and Tiny YOLO [198]; and (c) mixed weight Workload-C (from request to all the nine DNNs). To generate multi-tenant instances from these scenarios, we assign a random arrival time for each request from a Poisson distribution, commensurate with MLPerf and other works [255, 240, 130] to mimic task dispatching in datacenters. We assign priority levels within the range of 1 to 11 according to [175] to the dispatched tasks from a uniform distribution. We use Quality of Service (QoS) constraints presented by MLPerf for the server scenarios. To well exercise our proposed system, we use three levels of QoS for each workload scenario, (a) QoS-S as a soft QoS constraint (defined as $1\times$ QoS given in MLPerf), (b) QoS-M as a medium constraint ($\frac{1}{4}\times$ QoS), and (c) QoS-H as a hard constraint ($\frac{1}{16}\times$ QoS) to evaluate sensitivity to QoS latency constraints.

Hardware modeling. We implement the proposed omni-directional systolic subarray and the bussing systems including crossbars for the Fission Pod s in Verilog and synthesize them with Synopsys Design Compiler (L-2016.03-SP5) using FreePDK-45nm standard cell library [10] to extract their power/area. We model the on-chip SRAM using CACTI-P [155] that provides energy and area. The on-chip busing system is modeled using McPAT 1.3 [156] and the energy cost estimated to be 0.64 pJ/bit per hop.

Simulation infrastructure for Planaria. We compile each DNN benchmark to Planaria, and develop a cycle-accurate simulator that provides the cycle counts and statistics for energy measurements for each DNN using the modeling described above. We include all the overheads of reconfiguration, fission, instruction fetch, off-chip memory accesses, etc. We verify the cycle counts with our Verilog implementations.

Comparison with PREMA. We compare our proposed Planaria accelerator that supports spatial multi-tenant execution of DNNs to PREMA [62] that offers multi-tenancy via temporal execution. Baseline PREMA utilizes a monolithic TPU-like systolic DNN accelerator as its hardware. For fair comparison, we use the same number of PEs (128×128=16,384), on-chip activation/weight/output buffers (12 MB), frequency (700 MHz), and off-chip memory bandwidth as reported in PREMA. The detailed analysis of the synthesis results shows that our design can meet 1GHz frequency and the added omni-directional links or the buses are not on the critical path due to pipelining. However, for fair comparison with PREMA [62], we still use their reported 700 MHz frequency which is based on TPU [128].

PREMA's monolithic systolic array is not explicitly optimized to execute the most recent DNNs that use depth-wise convolutions such as EfficientNet and MobileNet. For this reason, in our evaluation, Workload-A does not include any DNNs with separable depth-wise convolutions. However, it is most reasonable to expect both heavy and lightweight workloads running on the same accelerator, according to industry collaborators. For instance, Google Photos runs image classification (e.g., GoogLeNet), object detection (e.g., MobileNet), and text recognition (e.g.,

GNMT) all on the same accelerator. One of Planaria's non-tangible benefits is its adaptability to various DNNs that is not available in monolithic designs. Moreover, the "light" and "heavy" are merely MLPerf terminologies. Even light benchmarks such as MobileNet-v1 require 1.1 billion operations/4.2 million parameters. We, in good faith, segregate these DNNs to eliminate any bias in our comparisons.

Evaluation Metrics. To evaluate the effectiveness of the proposed solutions, we use the following metrics:

• **Throughput** is defined as the maximum queries-per-second ($\frac{1}{\lambda}$) achieved by the system according to the Poisson distribution (λ) while meeting the SLA for different QoS constraints (QoS-S, QoS-M, and QoS-H). According to MLPerf [197], meeting SLA is defined as executing an image classification or object detection task 99% of the time and a translation task (e.g. GNMT) 97% of time within its QoS latency bound in a multi-tenant workload. This is the main metric for evaluation of server scenarios for inference tasks in MLPerf [197].

• **SLA Satisfaction Rate** is the fraction of multi-tenant workloads that adheres to the SLA described above.

• **Fairness** measures the equal progress of the tasks while considering task priorities. We use the same definition for fairness given in PREMA baseline [62], as: $fairness = min_{i,j}\frac{PP_i}{PP_j}$, while $PP_i = \frac{T_i^{isolated}}{T_i^{multi-tenant}} / \frac{Priority_i}{\sum Priority_k}$.

• **Energy reduction** compares total energy consumption to run multi-tenant workloads on both Planaria and PREMA.

5.6.2 Experimental Results

Comparison with PREMA

Throughput comparison. Figure 5.12 compares the throughput of Planaria with PREMA across various workload scenarios and QoS requirements. For Workload-C as the most comprehensive workload scenario that encompasses all the benchmark DNNs, Planaria improves the throughput by 7.4×, 7.2×, and 12.2×, for QoS-S, QoS-M, and QoS-H, respectively. For Workload-B the

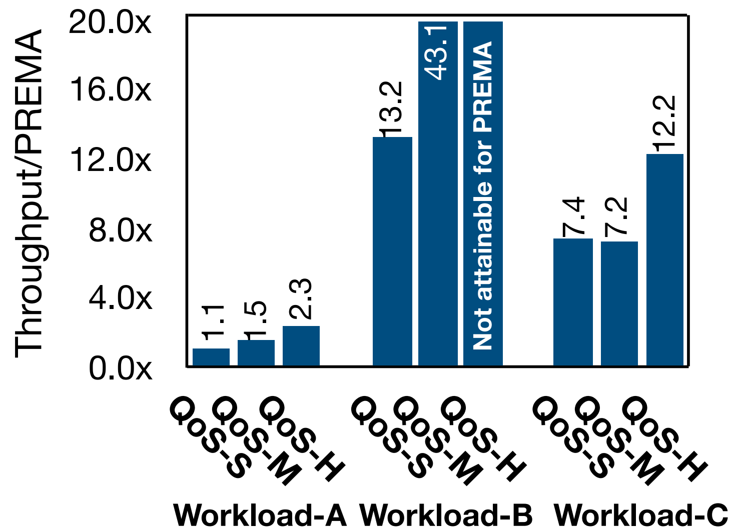


Figure 5.12. Throughput improvement over PREMA.

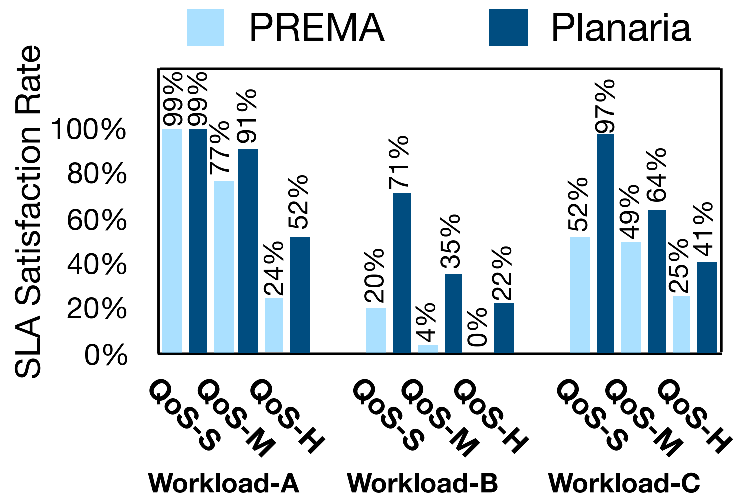


Figure 5.13. SLA satisfaction rate comparison.

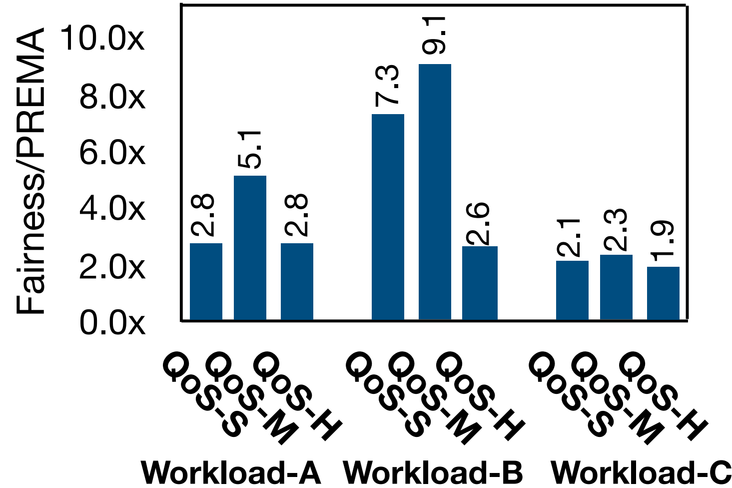


Figure 5.14. Fairness improvement over PREMA.

improvements increase to 13.2× and 43.1× for QoS-S and QoS-M, respectively, while for QoS-H, the baseline PREMA does not meet the 99% QoS constraints. This trend emanates from the fact that the DNNs in Workload-B include separable depth-wise/point-wise convolutions (except for Tiny YOLO). Since Planaria has fission capability, it can better utilize its resources for depth-wise convolution while a monolithic design in PREMA cannot conform to the requirements of this layer. This is an additional advantage of fission that enables running these recent DNNs more efficiently. With regard to Workload-A, Planaria improves the throughput by 1.1×, 1.5×, 2.3×, for QoS-S, QoS-M, QoS-H, respectively. These DNNs do not include depth-wise convolution, yet our hardware and scheduling yields significant benefits. Across all three workload scenarios, improvements are more significant for the case of hard QoS. Planaria performs better than PREMA in meeting the stricter QoS requirements, as its scheduler is QoS-aware and allocates resources to tasks based on their QoS.

SLA satisfaction rate comparison. Figure 5.13 illustrates the SLA satisfaction rate of Planaria and PREMA for a the same throughput ($\frac{1}{\lambda}$). As the results show, Planaria improves the SLA satisfaction rate across all the workloads and QoS requirements. The Planaria's *fission-capable* microarchitecture combined with its *QoS-aware* task scheduling algorithm enables significantly larger number of workloads to be executed while adhering to SLA, compared to PREMA.

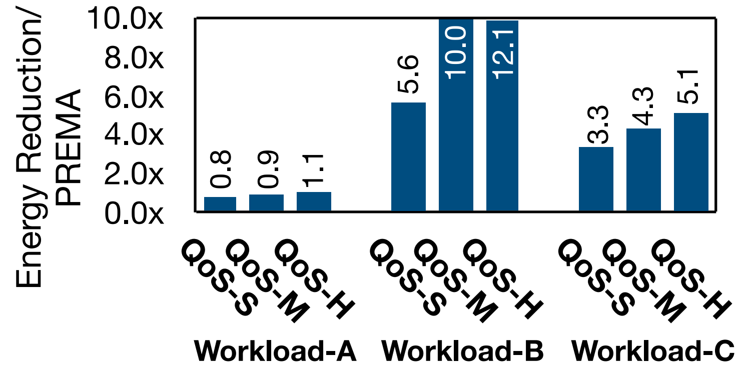


Figure 5.15. Planaria energy reduction compared to PREMA.

Based on the adopted QoS constraints from [197], Workload-A allows relatively larger slack time compared to other workloads. As such, both Planaria and PREMA performs relatively better in SLA satisfaction for Workload-A. Except for the case of QoS-S, where both Planaria and PREMA satisfy the SLAs 99% of the time, Planaria provides a 14% and 28% increase in SLA satisfaction rate compared to PREMA. For the case of Workload-B that requires tighter QoS as compared to Workload-A, improvements increase to 22%, 31%, and 51%, for QoS-H, QoS-M, and QoS-S, respectively. Finally, the improvements ranges from 16% to 45% for QoS-S to QoS-H, with respect to the mixed Workload-C.

Fairness comparison. Figure 5.14 shows fairness with Planaria normalized to fairness with PREMA across all the three workload scenarios. Planaria significantly improves fairness for Workload-A by 2.8×, 5.1×, 2.8× across the three QoS requirements. Overall, Planaria improves fairness significantly with minimum of 1.9× for (Workload-C, QoS-H) and maximum of 9.1× for (Workload-B, QoS-M). That is because spatial co-location in Planaria allows multiple tasks to progress simultaneously, whereas in temporal co-location only one task is privileged to be executed at a time. Besides, spatial co-location takes advantage of existing underutilized resources to improve execution of other tasks. In addition to that, Planaria's task scheduling algorithm (functions ALLOCATEFITTASKS and ALLOCATEUNFITTASKS in Algorithm 1) ensures that each dispatched task receives adequate number of subarrays with respect to its priority and overall execution time.

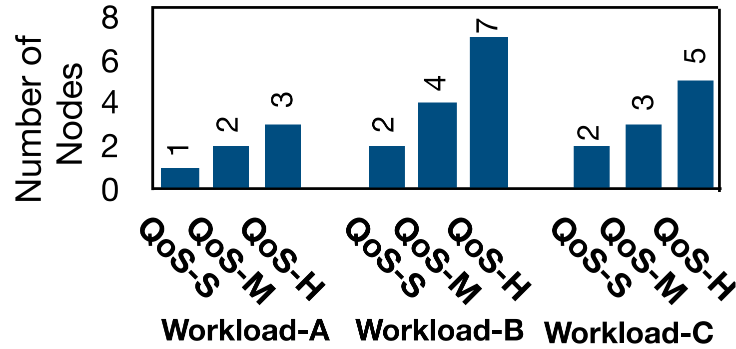


Figure 5.16. Required number of nodes to achieve 99% SLA satisfaction. PREMA is not designed for SLA. To avoid unfairness, the results for PREMA is omitted.

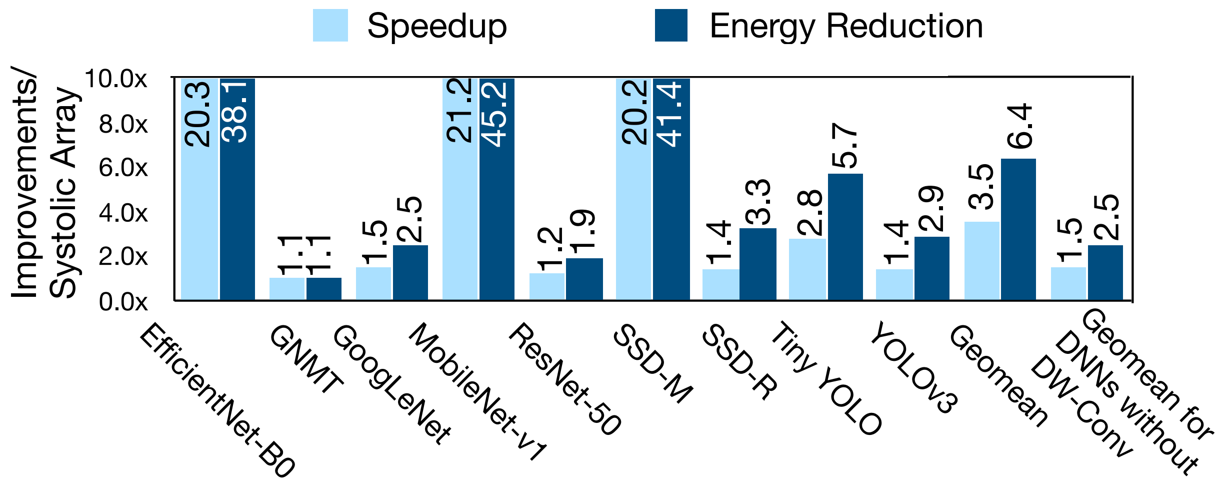


Figure 5.17. Planaria improvements for single DNN inference compared to a conventional systolic accelerator with the same on-chip memory and compute resources.

Energy comparison. Figure 6.10 compares the total energy consumption for the execution of workloads on Planaria and PREMA systems. For Workload-A, Planaria consumes slightly more energy than PREMA ranging 11% (QoS-M) to 25% (QoS-S). Multi-tenancy leverages the slack in QoS requirements and as such runs the application slightly slower than an isolated mode to improve throughput and fairness. This slower execution manifests itself as increased total energy compared to running each DNN in isolation with fastest possible speed without considering QoS. As a result, we see a degree of total energy increase for these traditional workloads. In the case of Workload-B and Workload-C, however, when modern DNNs are mixed, the energy benefits from fission outweighs this effect. Workload-B enjoys the maximum energy improvements using Planaria, with minimum of 5.6× and maximum of 12.1× gains over PREMA. A subset of the DNNs in Workload-B require depth-wise convolution layers. Planaria's fission capability significantly reduces the underutilization that monolithic systolic designs suffer due to these layers. Hence, this increase in utilization leads to a higher speedup and lower energy consumption for this workload. More details are presented in Section 5.6.2. Overall, with respect to Workload-C which is a mixture of both DNN classes, Planaria reduces the total energy consumption of the workloads by 3.3×, 4.3×, and 5.1× for QoS-S, QoS-M and QoS-H, respectively.

Scaling out resources. Figure 5.16 illustrates the minimum number of Planaria nodes necessary to achieve 99% SLA satisfaction, using a constant throughput across all workloads and QoS requirements. In this scaled-out setting, the DNN task traffic is distributed across multiple Planaria-equipped node, where each node has one accelerator. Each DNN task is mapped to a single ship instead of being distributed across multiple nodes. As illustrated in the figure, the number of nodes necessary to achieve SLA satisfaction increases as we go from soft (QoS-S) to hard constraints (QoS-H) on QoS. Among the workload scenarios, Workload-B, which has stricter QoS constraints, requires larger number of nodes compared to other workloads, with minimum of 2 nodes for QoS-S and maximum of 7 nodes for QoS-H. Also, one Planaria accelerator is sufficient for QoS-S of Workload-A which already satisfies the SLAs 99% of the time (Figure 5.13)

and thus obviates the need for an increased number of nodes, whereas QoS-H requires three nodes. Finally, with regard to Workload-C, 2, 3, and 5 nodes are required for QoS-S, QoS-M, andQoS-H, respectively.

Sensitivity Studies

Planaria performance/energy on a single DNN inference. Figure 5.17 shows the speedup and energy reduction of Planaria as compared to a conventional systolic-based accelerator (similar to PREMA's) with the same amount of compute and memory resources, while each DNN inference is executed in isolation. Across the nine DNN benchmarks, Planaria offers 3.5×and 6.3× speedup and energy reduction, respectively. The fission-capable design of Planaria enables it to adapt to the various computational characteristics that exist in DNN layers and exploits the opportunities for parallelism and data reuse to improve the performance and energy consumption.

Among them, EfficientNet-B0, MobileNet-v1, and SSD-M which exploit depth-wise convolutions enjoy the maximum benefits. To run depth-wise layers on monolithic systolic arrays, a depth-wise 2-D filter is vectorized and mapped to one column of the array. Lack of input reuse in depth-wise convolution leads to utilizing only one column of the array. This column then accumulates the results of the multiplication of depth-wise filter and its corresponding inputs while the filter weight remains stationary for all the inputs of the pertinent channel. Architecture fission capability and dynamic reconfigurability of Planaria enables it to fission into 16 *independent* smaller subarrays for executing the depth-wise layers. As such, 16 systolic columns, each of which from different subarrays, are utilized to process 16 channels in *parallel* for depth-wise convolution, yielding up to 16× higher utilization. Therefore, the proposed architecture fission yields significantly higher performance for depth-wise convolution.

With regard to DNNs without depth-wise convolution, Tiny YOLO achieves the maximum benefits, 2.8× speedup and 5.5× energy reduction. GNMT attains the least improvements, since it mostly requires matrix-multiplication operations, which is also suitable for a monolithic design. Unlike the multi-tenant case for Workload-A, there is no increase in energy for its isolated DNNs.

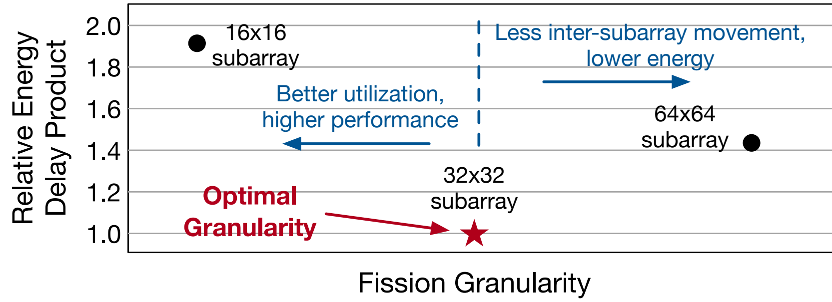


Figure 5.18. Design space exploration for fission granularity.

Table 5.2. Layer sensitivity to various fission configurations. Each cell shows a configuration with its architectural attributes (parallelism, input activation reuse, partial sum reuse, and usage of omni-directional data movement) and the percentage of the layers that uses the configuration.

A 32x32 Omni-diractional Systolic Subarray P: Parallelism IAR: Input Activation Reuse PSR: Partial Sum Reuse OD-SA: Omni-Directional Systolic Array Feature

Configuration	P	IAR	PSR	OD-SA	DNN	% of Layers
(128x128)-1 Cluster	1x	4x	4x	Unused	EfficientNet-B0	7.3 %
					GoogLeNet	15.5 %
					MobileNet-v1	7.1 %
					ResNet-50	26.5 %
					SSD-M	26.1 %
					SSD-R	62.5 %
					Tiny YOLO	11.1 %
					YOLOv3	21.2 %
(32x512)-1 Cluster	1x	16x	1x	Used	EfficientNet-B0	15.9 %
(512x32)-1 Cluster	1x	1x	16x	Used	EfficientNet-B0	9.8 %
					GoogLeNet	15.5 %
					ResNet-50	6.1 %
					Tiny YOLO	22.2 %
					YOLOv3	7.7 %
(64x256)-1 Cluster	1x	8x	2x	Used	EfficientNet-B0	11.0 %
					GoogLeNet	1.7 %
					ResNet-50	8.2 %
(256x64)-1 Cluster	1x	2x	8x	Used	EfficientNet-B0	7.3 %
					GNMT	100 %
					GoogLeNet	29.3 %
					MobileNet-v1	35.7 %
					ResNet-50	32.6 %
					SSD-M	26.1 %
					SSD-R	16.7 %
					Tiny YOLO	22.2 %
					YOLOv3	53.8 %
(32x256)-2 Clusters	2x	8x	1x	Used	EfficientNet-B0	2.4 %
					YOLOv3	1.9 %
(256x32)-2 Clusters	2x	1x	8x	Used	EfficientNet-B0	1.2 %
					GoogLeNet	12.1 %
					Tiny YOLO	11.1 %
(64x128)-2 Clusters	2x	4x	2x	Unused	GoogLeNet	6.9 %
					MobileNet-v1	3.6 %
					SSD-M	4.3 %
					SSD-R	4.2 %
					YOLOv3	5.8 %
(128x64)-2 Clusters	2x	2x	4x	Unused	GoogLeNet	3.4 %
					ResNet-50	12.2 %
					YOLOv3	5.8 %
(32x128)-4 Clusters	4x	4x	1x	Unused	EfficientNet-B0	1.2 %
					SSD-M	8.7 %
(128x32)-4 Clusters	4x	1x	4x	Unused	GoogLetNet	8.6 %
					SSD-M	4.3 %
(64x64)-4 Clusters	4x	2x	2x	Unused	EfficientNet-B0	1.2 %
					ResNet-50	8.2 %
					SSD-R	16.7 %
					Tiny YOLO	11.1 %
					YOLOv3	1.9 %
(64x32)-8 Clusters	8x	1x	2x	Unused	GoogLeNet	1.7 %
					Tiny YOLO	11.1 %
					YOLOv3	1.9 %
(32x64)-8 Clusters	8x	2x	1x	Unused	EfficientNet-B0	2.4 %
					MobileNet-v1	7.1 %
					SSD-M	4.3 %
(32x32)-16 Clusters	16x	1x	1x	Unused	EfficientNet-B0	23.1 %
					GoogLeNet	1.7 %
					MobileNet-v1	46.4 %
					ResNet-50	6.1 %
					SSD-M	26.1 %
					Tiny YOLO	11.1 %

As discussed, multi-tenancy trades off individual energy and speed for higher throughput. In the isolated case, that trade-off is not employed and all the resources are allocated to one DNN maximizing its efficiency and speed through fission.

Design space exploration for fission granularity. To find the optimal fission granularity, we perform a design space exploration that yields the most efficient granularity, as shown in Figure 5.18. We consider 128×128 total number of PEs (as was in PREMA[62] and TPU [128]) and sweep the size of subarrays for 16×16, 32×32, and 64×64. To find the optimal size, we consider Energy-Delay-Product (EDP) and measure its average value across the benchmark DNNs, while they run in isolation. Figure 5.18 illustrates the relative EDP values for the three design points. Blue arrows in Figure 5.18 show the tradeoff between energy and performance for the design space exploration with respect to the fission granularity. As Figure 5.18 shows,

32×32 PEs per subarray offers least EDP, that considers both energy and performance. This is the size that Planaria has adopted for its fission granularity.

Sensitivity analysis for fission possibilities. Table 5.2 illustrates the DNN layers sensitivity to various fission possibilities, where the whole accelerator is dedicated to one DNN inference. The dark blue cells of the table show the 15 most fitting fission possibilities for the benchmarks when run in isolation. The table also reports their architectural characteristics (parallelism (P), input activation reuse (IAR), partial sum reuse (PSR), and usage of omni-directional systolic movement (OD-SA)) with respect to the 32×32 fission granularity. A cell also lists the DNNs with the percentage of their layers that have utilized the pertinent fission configuration. Omni-directional systolic design enables six of these configurations. The black cell in Table 5.2 captures the most prevalent and fruitful fission configuration that, in fact, exploits the omni-directional feature. All nine DNNs utilize this configuration in their execution, where GNMT, YOLOv3, and MobileNet-v1 are the three DNNs that utilize this configuration more than others. Another important configuration is where fission takes place at the finest granularity and 16 of 32×32 subarrays work independently in parallel. This configuration is specifically important and useful for DNNs with depth-wise convolution, e.g. EfficientNet-B0, MobielNet-v1.

Area and power overheads for fission. Figure 5.19 illustrates the breakdown of area and power with respect to different hardware components in Planaria when synthesized at 45 nm, without considering on-chip buffers that are the same as one used in PREMA. The breakdown includes the components added to support dynamic fission, which includes the logic for Omni-directional flow of data, Fission Pod crossbar, SIMD vector unit additions, instruction buffer additions, and re-configurations registers. Other components are multipliers, adders and accumulators, pipelining registers for intra-systolic array/subarray data movement, a SIMD unit, control logic, and an instruction buffer. Note that these components are the same for both regular systolic array and Planaria, and consequently these are not considered overheads. Overall, dynamic fission adds 12.6%, 20.6% extra area and power, respectively.

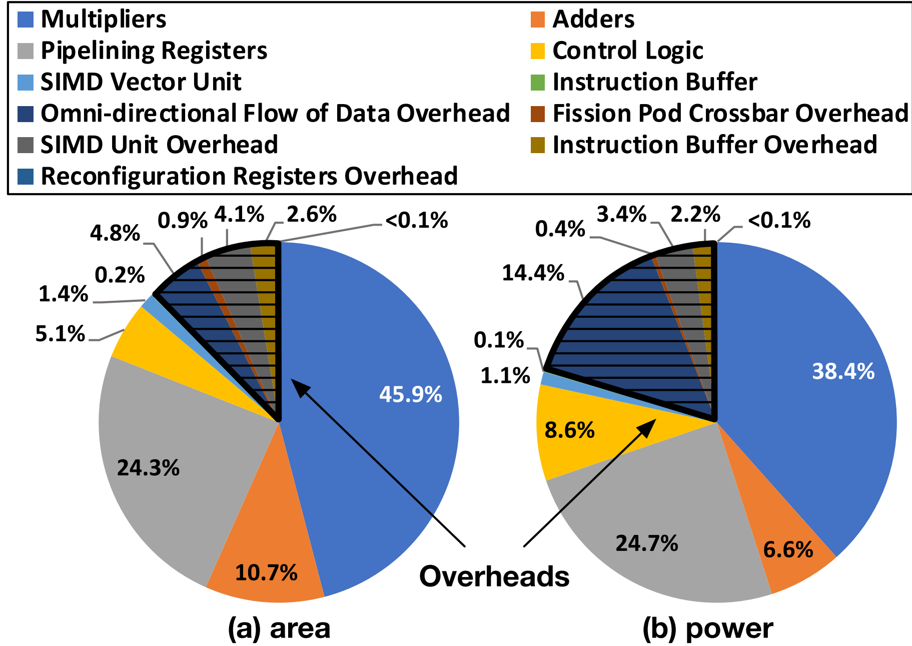


Figure 5.19. Planaria power/area breakdown and its overheads.

5.7 Related Work

The need for higher speed and efficiency in DNN execution has led to an explosion of DNN accelerators [58, 55, 106, 29, 161, 295, 129, 215, 59, 136, 196, 92, 30, 221, 184, 128, 227, 28, 222, 27, 90, 112, 152, 74, 219] that has even made their way to operational datacenters (Google's TPU [128], NVIDIA T4 [17], Microsoft Brainwave [90], etc.). However, multi-tenancy has been largely omitted in the proposed or deployed designs due to the arms race in the market for higher speed and efficiency. This paper offers spatial multi-tenant acceleration through architecture fission that is propelled by unique microarchitectural mechanisms and organizations that enables flexible task scheduling. As such, this paper lies at the intersection of DNN acceleration and multi-tenant execution. We discuss relevant related work categories below.

Multi-tenancy for DNN accelerators. PREMA [62] develops a scheduling algorithm for preemptive execution of DNNs on a monolithic accelerator and uses time-sharing for multi-tenancy. On the other hand, AI-MT [36] develops an architecture that supports multi-tenancy by

first tiling the layers at compile time, then exploiting hardware-based scheduling to maximize resource utilization. In contrast to these temporal multi-tenancy supports, this paper explores architecture design for spatial co-location of DNNs for multi-tenant acceleration and its unique scheduling challenges.

Flexibility in DNN accelerators. Flexibility in DNN acceleration has recently gained attention [57, 191, 147, 222, 216, 94]. However, these inspiring works do not explore simultaneous spatial co-location of multiple DNNs on the same chip. Eyeriss v2 [57] proposes a hierarchical architecture equipped with a flexible mesh-based NoC that provides flexibility to adapt to various level of data reuse. MAERI [147] and SIGMA [191] propose a reconfigurable interconnect among the PEs to deal with sparsity in neural networks [147] and matrix multiplications [191] and to increase resource utilization. Simba [216] proposes a scalable multi-chip module-based accelerator to reduce fabrication cost and provide scalability with respect to inter-chip and intra-chip communication. BitFusion [222] explores bit-level dynamic composability in its multipliers to support heterogeneity in deeply quantized neural networks. Tangram [94] explores dataflow optimizations by buffer-sharing dataflow and inter-layer pipelining on a hierarchical design to reduce energy.

Multi-tenancy for CPUs and GPUs. There is a large swath of related work on multi-tenancy for CPUs [226, 246, 171, 172, 263, 71, 296, 70, 130, 251, 250, 100, 99] and GPUs [245, 186, 187, 300, 265, 213, 50, 49, 88, 254, 160, 259, 47, 130, 195, 40] due to its vitality for cloud-scale computing. NVIDIA Triton Inference Server [19] (formerly TensorRT Inference Server) provides a cloud software inference solution optimized for GPUs and offers benefits by supporting multi-tenant execution of DNNs on them [225]. GrandSLAm [130] proposes scheduling policies to minimize SLA violation rates for microservices in the cloud for CPUs and GPUs, and the studied workloads include DNNs. In contrast, this paper uniquely enables spatial multi-tenancy on DNN accelerators, by leveraging a dynamic fission in the architecture and leveraging that through the scheduler. Kubernetes [14] and Mesos [114] are cloud-scale resource management framework,

but have not explored spatial multi-tenancy in DNN accelerators due to its unavailability. Our scheduling algorithm is complementary to their operation.

DNN acceleration. There is a large body of work [58, 55, 221, 128, 90, 94, 57, 216, 146, 106, 29, 161, 295, 184, 28, 147, 74, 219, 191, 129, 196, 222, 220, 97, 98, 26, 25, 206, 279, 203, 211, 118, 152, 80, 136, 215, 59, 227, 92] for isolated acceleration of DNNs that, although inspired our work, are not focused on multi-tenancy but rather offer various innovations to improve the speed and efficiency of DNN execution.

5.8 Conclusion

As INFerence-as-a-Service is growing in demand, it is timely to explore multi-tenancy for DNN accelerators. This paper explored this topic through a novel approach of dynamic architecture fission, and provided a concrete architecture, Planaria, and its respective scheduling algorithm. Evaluation with a diverse set of DNN benchmarks and workload scenarios shows significant gains in throughput, SLA satisfaction, and fairness.

5.9 Acknowledgement

Chapter 5 is a partial reprint of the material as it appears in: S. Ghodrati, B. Ahn, J. Kim, S. Kinzer, B. Yatham, N. Alla, H. Sharma, M. Alian, E. Ebrahimi, N. Kim, C. Young, and H. Esmaeilzadeh, "Planaria: Dynamic Architecture Fission for Spatial Multi-Tenant Acceleration of Deep Neural Networks." in *International Symposium on Microarchitecture (MICRO)*, 2020. The dissertation author was the primary investigator and author of this paper.

Chapter 6

Leveraging Learning Algorithms to Maximize Execution Efficiency of Transformer Models

6.1 Introduction

Natural Language Processing (NLP) defines the frontier for Artificial Intelligence (AI) as it is even central to the Turing Test [23]. The recent advent of the self-attention mechanism [256] enabled unprecedented successes in the field of NLP, shifting the focus of deep learning from convolutional neural networks towards Transformer models in various domains [133, 69, 134, 148, 193, 163, 285, 60, 257]. The self-attention mechanism calculates a score to measure the correlation between a word and all the other words in a subtext. The subtext is the collection of all the words, which is captured by the attention mechanism. Therefore, it quantifies the context of the word under attention with respect to its subtext.

Intuitively, a word can bear multiple connotations, of which only one is expressed in its proximate context. Usually, few keywords define the context and therefore, a significant amount of computation will be inconsequential. The attention score for a word determines highly correlated words; the rest are merely irrelevant. There exists a threshold that differentiates between the scores of the words that need to be considered and those that do not define the context and are thus inconsequential. Because each attention layer identifies a distinct context

of the target sentence, such a threshold needs to be defined on a per-layer basis to maintain model accuracy. Recent research has leveraged this insight and proposed several techniques that skip computation if a threshold is not met [261, 104, 243, 103]. Clearly, skipping computation negatively impacts model accuracy, which is also dependent on the value of the thresholds. Therefore, establishing the right thresholds is crucial for the efficacy of the runtime computation pruning methods. However, the literature [261, 104, 243, 103] has relied on heuristics, statistical sampling, or human input that do not provide reliable expected accuracy.

In contrast, this paper formulates the problem of finding thresholds for the attention layers as a regularizer that amends the loss function of the transformer model. Our technique is robust even though the threshold values are discrete and cannot be directly optimized through gradient-based approaches. *A key contribution of this work is to formulate finding the layer-wise pruning thresholds as a differentiable regularizer. This formulation leverages the gradient-based back-propagation algorithm to mathematically co-optimize the threshold values and weight parameters.* This approach unblocks simultaneous co-optimization of the two conflicting objectives of maximizing the pruning rate of the computations while minimizing the accuracy loss. In addition, this analytical technique strikes a formally optimal balance between accuracy and computation pruning. Note that the current Cambrian explosion of deep learning hinges upon two main algorithmic innovations. First, changing the activation function of perceptrons from a non-differentiable step function to the continuous smooth sigmoid function [205] enabled back-propagation and multi-layer neural networks and ended the first AI winter [21]. Second, solving the vanishing gradients problem [140] has resulted in stable training deep neural networks that have taken the IT industry by storm. The proposed approach is analytical and therefore mathematically sound, and does not rely on limited empirical evidence. The solution also guarantees the same generality and optimality that are essential for training the machine learning model itself.

In this paper, we apply these algorithmic innovations to learn self-attention thresholds in a gradient-based fashion. At runtime, the attention scores below the learned threshold are replaced

by $-\infty$ to void their impact on the attention layer's outputs. As such, the preceding computation can be pruned early when the result is below the threshold. We devise a bit-serial architecture, called LEOPARD[1], to maximize the benefits by terminating computation even before pruning the following calculation. This design reduces computation at the finest granularity possible (bit level), hence offering benefits beyond pruning. Our hardware mechanism for early termination is exact and does not cause any accuracy degradation. *At the bit level, it can be determined ahead of calculation completions if the partial result of the dot-product can ever exceed the threshold. Therefore, another advance of this paper is leveraging arithmetic insights for early termination in the microarchitecture without approximation.*

We evaluate the effectiveness of our gradient-based algorithmic innovation and the proposed bit-level arithmetic properties by designing and implementing the LEOPARD accelerator in hardware. We synthesize and generate layout for a prototype of the LEOPARD accelerator implementation in a 65 nm process technology and characterize its speed and energy consumption under various settings. We evaluate various state-of-the-art transformer models, including BERT, GPT-2 and Vision-Transformer, and datasets forming a benchmark suite of 43 language and vision processing tasks. On average, the designed accelerator offers 1.9× and 3.9× speedup and energy reduction, respectively, compared to a baseline design without pruning and bit-level early termination support under an iso-area setting. LEOPARD's notable pruning rate can unlock more benefits, if more chip area budget (15%) is available. Given this extra area budget, our accelerator's benefits increase to 2.4×and 4.0×speedup and energy reduction, respectively. To better understand the sources of these improvements, we also distinguish between the effects of runtime computation pruning and bit-level early termination on energy savings. Our study across the target models shows that, on average, out of the 3.9× energy reduction, 2.1× stems from runtime computation pruning and 1.8× emerges from bit-level early termination. We also compare LEOPARD to two state-of-the-art accelerators for self-attention mechanism, A^3 [103] and SpAtten [261], which support runtime pruning. However, neither accelerator provides

[1]**LEOPARD**: **L**earning thr**E**sholds for **O**n-the-fly **P**runing **A**cceleration of t**R**ansformer mo**D**els.

analytical support or guarantee for model accuracy, only relying on heuristic approximations. The results from our evaluations suggest that formulating runtime pruning as a gradient-based optimization can unlock significant benefits, while guaranteeing inference accuracy.

6.2 Background and Motivation

6.2.1 Self-Attention Mechanism

"*Self-attention*" is a mechanism to find the relation between a word to all the other words in a sentence [256, 63]. To compute this relation, we first project each word to a vector with d_w dimensions, so-called embedding. Given a sentence with s words, this projection creates a matrix $\mathcal{X}$ with $s \times d_w$. Then, these word embeddings are multiplied into query weight matrix ($W^{\mathcal{Q}}$), key weight matrix ($W^{\mathcal{K}}$), and value weight matrix ($W^{\mathcal{V}}$), each with $d_w \times d$ dimensions as follows:

$$\mathcal{Q}_{s\times d} = \mathcal{X} \times \mathcal{W}^{\mathcal{Q}}; \quad \mathcal{K}_{s\times d} = \mathcal{X} \times \mathcal{W}^{\mathcal{K}}; \quad \mathcal{V}_{s\times d} = \mathcal{X} \times \mathcal{W}^{\mathcal{V}} \tag{6.1}$$

Given the query ($\mathcal{Q}$) and key ($\mathcal{K}$) matrices, a self-attention *Score* matrix is calculated as follows:

$$\text{Score}_{s\times s} = \mathcal{Q} \times \mathcal{K}^T \tag{6.2}$$

where each element s_{ij} in the self-attention *Score* matrix indicates the relation between word$_i$ and word$_j$ in the input sentence. The *Score* values are generally scaled down by $(\times 1/\sqrt{d})$ before the next step to enable stable gradients during training [256]. To ensure that the self-attention *Score*s are positive and adding up to one, "*softmax*" is applied to each row of Score matrix as follows:

$$\mathcal{P}_{s\times s} = \text{Softmax}(\text{Score}) \tag{6.3}$$

Softmax outputs indicate a probability estimation of the input words' relation. The self-attention values are calculated as follows:

$$\text{Att}_{s\times d} = \mathcal{P} \times \mathcal{V} \tag{6.4}$$

Generally, each attention layer consists of multiple heads each with dedicated $W^{\mathcal{Q}}$, $W^{\mathcal{K}}$, and $W^{\mathcal{V}}$ weight matrices. Each head presumably captures different dependencies between the token embeddings. In this case, the attention values (Equation 6.4) from each head are concatenated and projected into an attention matrix of size $s \times d_w$ using a weight matrix $\mathcal{W}^o_{(d\times h)\times d_w}$ as follows:

$$\text{Multi}-\text{Head Att}_{s\times d_w} = \text{Concat}(\text{Att}_1,\ \text{Att}_2,\ \cdots,\ \text{Att}_h) \times \mathcal{W}^o \tag{6.5}$$

where Concat operation concatenates the Att output matrix from each head to generate a $(s \times (d \times h))$-matrix.

6.2.2 Gradient-Based Optimization and Regularization

Gradient-based optimization. Training neural networks are formulated as an optimization problem of a predefined loss function. These loss functions are generally non-convex and have a manifold consisting of different local optima which makes the training of neural networks challenging. To alleviate the complexity of optimizing loss functions, it is common to use gradient-based methods [201, 138]. Using these gradient-based methods institute defining differentiable loss functions, such as cross-entropy [180] or Kullback-Leibler divergence [144] which is prevalent in self-attention models [134, 256, 238].

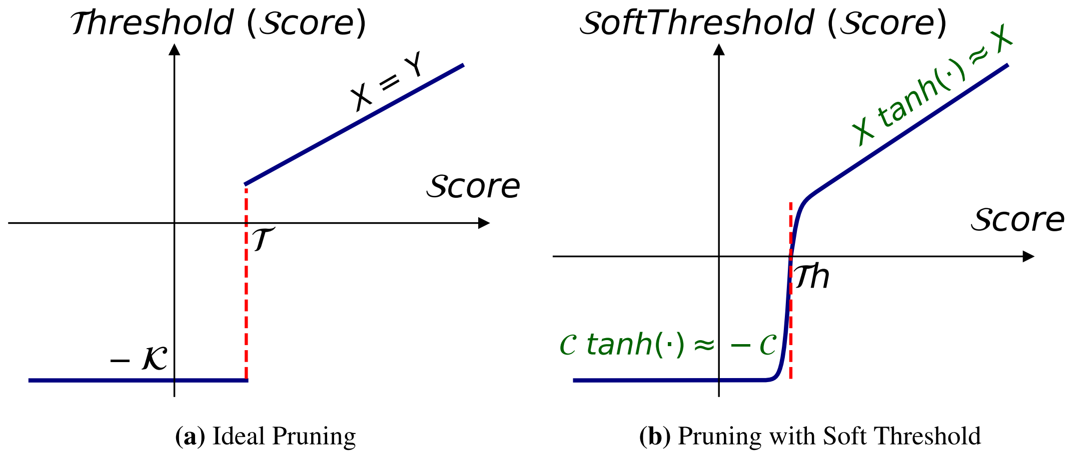
Regularization in loss function. To impose certain constraints on the model parameters, such as improved generalization [143, 304, 231] and introducing sparsity [91, 230, 35], it is common to use regularizer as part of the loss function. However, using gradient-based methods for training mandates these regularizers to be framed as additional differentiable terms to the loss. This differentiability constraint for employing gradient-based methods introduces a unique challenge for supporting constraints that are not inherently differentiable.

6.2.3 Motivation

Analyzing the computations for self-attention layers, it is apparent that the main computation cost is associated to Score (Equation 6.2) and attention computations (Equation 6.4) that necessitates the multiplications of two matrices with $s \times d$ dimensions, each with time complexity of $\mathcal{O}(s^2 d)$. These complexities translate to quadratic raise in computation cost and storage as the number of input tokens increases. As such, prior work aims to reduce the time and space complexity of these operations both from the algorithmic [63, 262, 42, 289, 60] and hardware perspectives [261, 104, **?**, 235, 103]. In this work, we propose an alternative pruning mechanism that learns the threshold as part of training. Our proposed technique prunes away unimportant *Score*s, hence eliminating the ineffectual computations of "*softmax(·)*" in Equation 6.3 and "$\times \mathcal{V}$" in Equation 6.4. In addition, to further cut down the computations of *Score*s (Equation 6.2), we employ a unique early-compute termination without impacting the model accuracy.

6.3 Algorithmic Optimizations for Sparse Attention

The section overviews the algorithmic optimizations for inducing sparsity in attention layers. We first introduce an online pruning method that eliminate *unimportant* attention layer computations as early as possible, right after *Score* calculations (e.g. $\mathcal{Q} \times \mathcal{K}^T$), to increase the realized performance benefits. Particularly, our method sets the layer-wise pruning thresholds as trainable parameters and jointly fine-tune the model parameters and learn the pruning thresholds as part of a light fine-tuning step. Then, our method compares the *Score* = $\mathcal{Q} \times \mathcal{K}^T$ values against the learned pruning thresholds per attention layer and prunes the ones that satisfy the pruning criteria. Note that, in contrast to prior learned weight pruning method for image classification models [35], the pruning criteria in our work is content-dependant and is applied adaptively based on the calculated *Score* values. That means the induced sparsity in attention layers by our approach varies from one content to another content. As our results indicate (See Section 6.5), the adaptive and content-dependant nature of our pruning method enables high sparsity in attention



(**a**) Ideal Pruning

(**b**) Pruning with Soft Threshold

Figure 6.1. Pruning operation on attention *Score*: (a) ideal magnitude-based pruning, (b) proposed differentiable pruning operation with soft threshold.

computations while yielding virtually no accuracy loss.

6.3.1 Learned Per-Layer Pruning

Learning per-layer pruning thresholds for attention layers consists of three main challenges. First, the search space of threshold values is complex and computationally intractable for exhaustive exploration. For example, BERT-L model has 24 layers creating a total of 24 threshold parameters, each of which can take any continuous value. Second, simply sweeping the threshold values as a one-time fine-tuning step could negatively affect the model accuracy [103, 261]. To mitigate these challenges, we propose to jointly fine-tune the model parameters and learn the threshold values as a light fine-tuning step with the joint objective of increasing model sparsity and retaining the baseline model accuracy. However, training the threshold values with the inherently non-differentiable pruning operation poses a unique challenge for gradient-based learned methods. For this, we use an approximate differentiable pruning operation and devise a surrogate regularizer to reinforce sparsity as part of the model loss function. In the following paragraphs, we expound our learned pruning method that couples two design principles, namely "*pruning with soft threshold*" and "*surrogate L_0 regularization*".

Pruning with soft threshold. Figure 6.1a demonstrates an ideal pruning operation for *Score*

values (e.g. $Score = \mathcal{Q} \times \mathcal{K}^T$, where $\mathcal{Q}$ and $\mathcal{K}$ are d-dimension vectors corresponding to a single word). The *Score* values greater than $\mathcal{T}h$ remain unchanged and those less than $\mathcal{T}h$ are clipped to a large negative number. As the pruning operation is followed by a "*softmax(·)*", setting the *Score* values below $\mathcal{T}h$ to a large negative number makes the output of the softmax operation sufficiently close to zero. Hence, the large negative numbers are pruned out of the following multiplication into $\mathcal{V}$. However, using this pruning operation as part of a gradient-based training method is not straightforward due to its discontinuity at $X = \mathcal{T}h$.

To circumvent the non-differentiality in the pruning operation, we propose to replace this operation with an approximate function that instead uses a soft threshold (shown in Figure 6.1b) as follows:

$$\text{SoftThreshold(x)} = \begin{cases} x\tanh(s(x-\mathcal{T}h)), & x \geq \mathcal{T}h \\ c\tanh(s(x-\mathcal{T}h)), & x < \mathcal{T}h \end{cases} \tag{6.6}$$

By assigning a reasonably large value to s, the shape of $tanh(\cdot)$ around $\mathcal{T}h$ becomes sharper and enables the learning gradients to effectively flow around this region. Supporting the learning gradients to flow at the vicinity of $\mathcal{T}h$ allow the gradient-based learning algorithm to either push down the model parameters (e.g. $\mathcal{Q}$ and $\mathcal{K}$) below the threshold or lift them above the threshold according to their contributions to the overall model accuracy.

Outside the vicinity of $\mathcal{T}h$, the $tanh(\cdot)$ asymptotically approaches one and the "Soft-Threshold" function simply becomes $\approx x$ and $\approx -c$ for values $\geq \mathcal{T}h$ and $< \mathcal{T}h$, respectively, which are close approximations of the original pruning operation. In our experiments, we empirically find that setting $c = 1000$ and $s = 10$ yield a good approximation for pruning and enables robust training.

Differentiable surrogate L_0 regularization. Using soft threshold as the sole force of pruning does not necessarily increase sparsity. Intuitively, the training method may just simply lower the threshold to be a small value, which translates to lower sparsity to maintain high model accuracy. Imposing such constraints to gradient-based methods are generally achieved through adding a

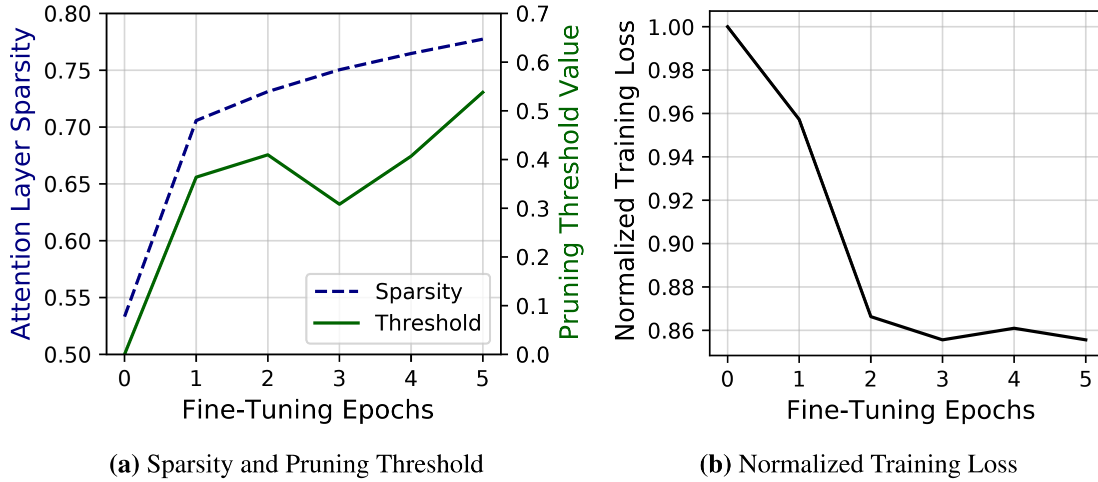
regularizer term to the loss function. A common method to explicitly penalize the number of non-zero model parameters is to use L_0 regularizer on model parameters in the loss function as follows: $L_{tot}(\theta) = \frac{1}{N}\left(\sum_{i=1}^{N} \mathcal{L}\left(A(x_i;\ \theta),\ y_i\right)\right) + \lambda ||\theta||_0$

$||\theta||_0 = \sum_{j=1}^{|\theta|} 1[\theta_j \neq 0]$ where $\mathcal{L}$ is the model loss function, $A(\cdot)$ is the model output for given input x_i and model parameters θ, y_i is the corresponding labeled data, λ is the balancing factor for L_0 regularizer, and 1 is the identity operator that counts the number of non-zero parameters.

Similar to "*Threshold*" function, L_0 regularizer suffers from the same non-differentiability limitation. To mitigate this, Louizos et al. [165] uses a reparameterization of model parameters to compute the training gradients. While this reparameterization technique yields state-of-the-art results for Wide Residual Networks [288] and small datasets, a recent study [91] shows that this reparameterization trick performs inconsistently for large-scale tasks such as attention models. In this work, we propose a simple alternative method that uses a differentiable surrogate L_0 regularization for the pruning of *Score* values in attention layers as follows: $||\theta||_0 = \sum_{j=1}^{|score|} 1[\text{score}_j > -c]$

$||\theta||_0 \approx \sum_{j=1}^{|score|} \text{sigmoid}(k(\text{score}_j + c - \alpha))$ where k = 100 and α = 1. Using these parameters forces the output of sigmoid($\cdot$) to asymptotically approach one for unpruned *Score* values and zero for the pruned ones, which are already bounded to $-c$ as shown in Equation 6.6. As such, the proposed differentiable surrogate L_0 regularizer is a close approximation of the original L_0 regularizer in Equation 6.3.1 (a).

Pruning mechanism. We apply our gradient-based learned pruning as a light fine-tuning step based on the previously proposed design principles: (1) pruning with soft threshold and (2) differentiable surrogate L_0 regularization. We employ the pre-trained attention models with the proposed modified loss function (e.g. original loss function and the surrogate L_0 regularizer) to jointly fine-tune the model parameters and learn the per-layer pruning thresholds. Using the proposed soft threshold mechanism in the fine-tuning step allows the gradient-based learning method to adjust the model parameters smoothly at the vicinity of the $\mathcal{T}h$ value. That is, pushing



(a) Sparsity and Pruning Threshold **(b)** Normalized Training Loss

Figure 6.2. An example (a) attention layer sparsity and its corresponding pruning threshold values and (b) normalized training loss as fine-tuning epochs progress for BERT-L model on QNLI task from GLUE benchmark.

down the non-important model parameters below threshold values and lifting up the important model parameters above it. One of the main benefits of using the proposed differentiable approach is enabling the model parameters to freely switch between prune and unpruned region. For all the studied attention models, we initialize the threshold values to zero and run the fine-tuning for up to five epochs.

Figure 6.2 demonstrates an example sparsity, threshold values, and normalized training loss curves for BERT-B model on QNLI task from the GLUE benchmark. Figure 6.2a shows that as fine-tuning epochs progress, both the sparsity and threshold values increase owing to the effectiveness of our joint co-training of sparsity and model parameters. The flexibility afforded by the joint co-training is further illustrated at the third epoch, where the sparsity continues to increase despite the corresponding decrease in the threshold value. Additionally, Figure 6.2b shows the decreasing trend of normalized training loss over the course of fine-tuning epochs.

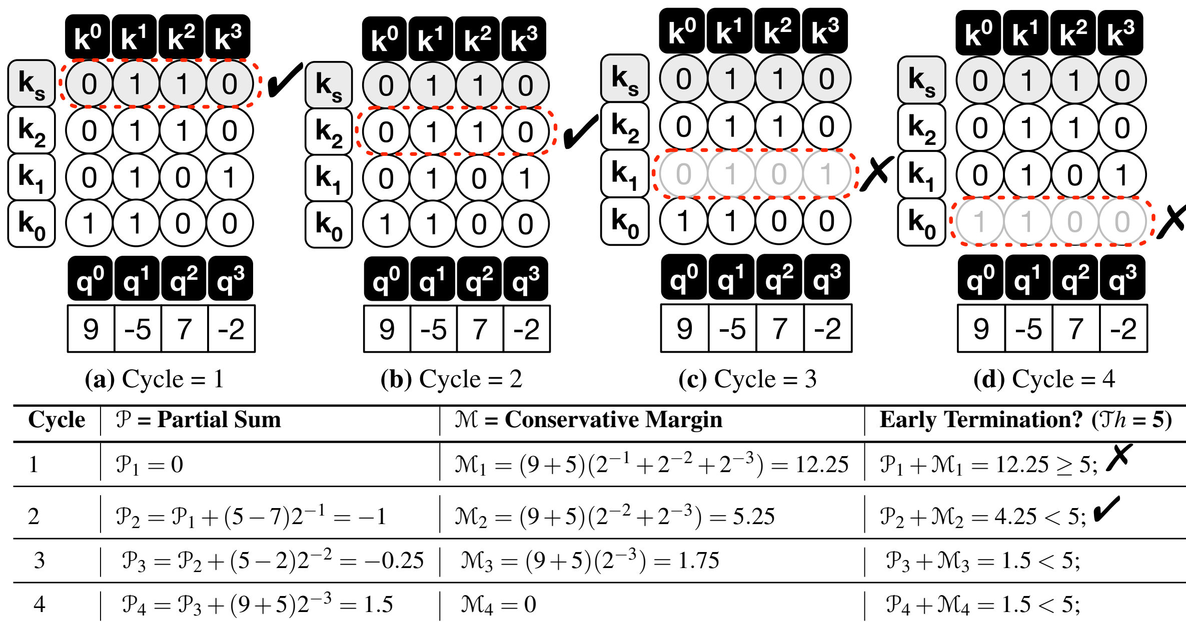
6.3.2 Bit-Level Early-Compute Termination

The learned pruning offers a unique opportunity to further improve the LEOPARD performance through bit-serial $\mathcal{Q} \times \mathcal{K}^T$ computation. If our system can anticipate that the

final result of $\mathcal{Q} \times \mathcal{K}^T$ computation is below the learned pruning threshold, the ongoing bit-serial computations can be terminated. However, this early-termination mechanism poses a key challenge in our design. As we desire to maintain the baseline model accuracy, the early-termination mechanism must not tamper with the computational correctness of attention layers. To address this, we propose to compute and add a dynamically adjusted conservative margin value to the partial sum during the bit-serial computations. The role of this margin is to account for the maximum potential increase in the remaining $\mathcal{Q} \times \mathcal{K}^T$ computations. If the addition of the partial sum values and the margin still falls below the learned pruning threshold value, the computations are terminated and the corresponding $\mathcal{Q} \times \mathcal{K}^T$ is simply pruned. In the following paragraph, we illustrate the proposed early-compute termination with a conservative margin.

Early-compute termination for dot-product operation. Figure 6.3 depicts the flow for a $\mathcal{Q} \times \mathcal{K}^T$ dot-product computation, each with four elements. $\mathcal{K}$ elements are placed in bit-serial format vertically from MSB $\rightarrow$ LSB, whereas $\mathcal{Q}$ values are stored in full-precision fixed-point format. In this example, the threshold value is set to five. For simplicity, we assume the computation is performed in sign-magnitude form, k_s represents the sign-bit for $\mathcal{K}$ vector, and the absolute values of $\mathcal{K}$ elements are less than one.

In the first cycle, the elements with concordant signs, (k^0, q^0) and (k^1, q^1), are used for margin initialization. The intuition here is that *only* the multiplications of elements with concordant signs can contribute positively to the final dot-product result. Multiplications of elements with opposing signs are ignored to keep the margin conservative and eliminate wrongful early compute terminations. As shown in the table of Figure 6.3, both the product of k^2 and the q vector as well as the margin are updated. The margin is adjusted to accommodate the largest possible positive contribution to the final value. In the second cycle, because the sum of $\mathcal{P}_2$ and $\mathcal{M}_2$ dips below the threshold, the computation process terminates. That is, the subsequent cycles (highlighted in gray) are no longer performed. Note that, with the proposed margin computation, we ensure that no approximation is introduced in the attention layers. Next section discusses the



Cycle	$\mathcal{P}$ = Partial Sum	$\mathcal{M}$ = Conservative Margin	Early Termination? ($\mathcal{T}h$ = 5)
1	$\mathcal{P}_1 = 0$	$\mathcal{M}_1 = (9+5)(2^{-1}+2^{-2}+2^{-3}) = 12.25$	$\mathcal{P}_1+\mathcal{M}_1 = 12.25 \geq 5$; ✗
2	$\mathcal{P}_2 = \mathcal{P}_1+(5-7)2^{-1} = -1$	$\mathcal{M}_2 = (9+5)(2^{-2}+2^{-3}) = 5.25$	$\mathcal{P}_2+\mathcal{M}_2 = 4.25 < 5$; ✓
3	$\mathcal{P}_3 = \mathcal{P}_2+(5-2)2^{-2} = -0.25$	$\mathcal{M}_3 = (9+5)(2^{-3}) = 1.75$	$\mathcal{P}_3+\mathcal{M}_3 = 1.5 < 5$;
4	$\mathcal{P}_4 = \mathcal{P}_3+(9+5)2^{-3} = 1.5$	$\mathcal{M}_4 = 0$	$\mathcal{P}_4+\mathcal{M}_4 = 1.5 < 5$;

Figure 6.3. High-level overview of early-compute termination for dot-product operation $\mathcal{Q} \times \mathcal{K}^T$. In this example, $\mathcal{K}$ is represented in bit-serial format, whereas $\mathcal{Q}$ is in full-precision fixed-point format. In Figure (a-d) each column illustrate one element of $\mathcal{K}$ vector and each row represents its corresponding bits (MSB -> LSB). K_s indicates the sign bit. For simplicity, $\mathcal{K}$ elements are scaled to be between -1.0 and +1.0. The table shows the partial sum values after each cycle.

hardware realization for this proposal.

6.4 LEOPARD Hardware architecture

We design LEOPARD hardware while considering the following requirements based on our algorithmic optimizations:

1. Leveraging the layer threshold values to detect the unpruned *Score*s and their corresponding indices in the output matrix.
2. Using bit-serial processing to early-stop the computation of pruned *Score*s and associated memory access .
3. Processing the $\times\mathcal{V}$ operation for only un-pruned *Score*s to minimize operations while achieving high compute utilization.

6.4.1 Overall Architecture

Due to abundant available parallelism in multi-head attention layers, we design a tile-based architecture for LeOPArd, where attention heads are partitioned across the tiles, and the operations in the tiles are independent of each other on their corresponding heads. Figure 6.4 illustrates the high-level microarchitecture of a single LeOPArd tile. Each tile comprises two major modules to process the computations of attention layers:

1. A front-end unit, dubbed Query Key Processing Unit (QK-PU), that streams in the $\mathcal{Q}$ vectors (row by row from the Q matrix, where each row corresponds to a word) from the off-chip memory, reads $\mathcal{K}$s from a local buffer, and performs vector-matrix multiplication between a $\mathcal{Q}$ vector and a $\mathcal{K}$ matrix. This unit also encompasses a 1-D array of bit-serial dot-product units, QK-DPU s, each of which equipped with logic to early-stop the computations based on the pruning threshold values and forward the unpruned *Score*s and their indices to the second stage.

2. A back-end unit, dubbed Value Processing Unit (V-PU), that performs softmax operations on the important un-pruned *Score*s to generate probability, and subsequently performs weighted-summation of the $\mathcal{V}$ vectors read from a local buffer to generate the final output of the attention layer.

The front- and back-end stages are connected to each other through a set of FIFOs that store the survived *Score*s and their corresponding indices. The front-end unit employs multiple (N_{QK}) QK-DPU s while sharing the single V-PU in consideration of high pruning rate during the processing in the front-end stage. If the front-end finished the computation with current $\mathcal{Q}$ vector, but the back-end is still working on the previous $\mathcal{Q}$ vector, the front-end unit is stalled until the completion of back-end unit. As the choice of N_{QK} is a key factor to maximize the overall throughput and back-end resource utilization, we explore this design space in Section 6.5.4, which leads to two choices of $N_{\text{QK}} = 6$ and 8 by focusing on area efficiency and higher utilization,

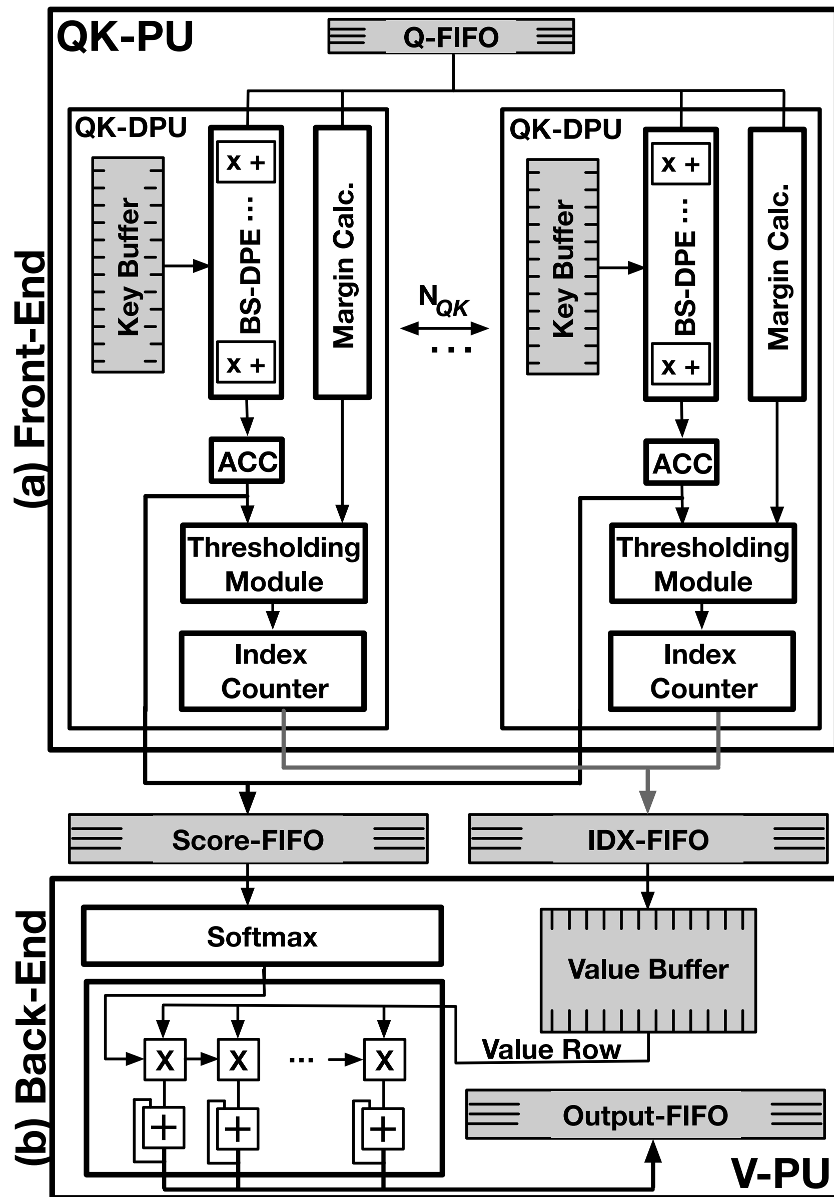


Figure 6.4. Overall microarchitecture of a LeOPArd tile.

respectively. Before the operation begins, all the $\mathcal{K}$ and $\mathcal{V}$ matrices are fetched from off-chip memory and stored on on-chip buffers, while the $\mathcal{Q}$ vectors are steamed in. Since the vectors are re-used by the number of sequence elements (e.g., 512 in BERT), DRAM costs are amortized.

6.4.2 Online Pruning Hardware Realization via Bit-serial Execution

As discussed in Section 6.3, to realize the pruning of redundant *Score*s during runtime and even earlier termination with bit-level granularity, we design LEOPARD front-end unit (depicted in Figure 6.4-(a)) as a collection of bit-serial dot-product units (QK-DPU).

Overall front-end execution flow. To perform the *Score* computations, the $\mathcal{Q}$ vectors are read sequentially from Q-FIFO and then broadcasted to each QK-DPU, while each QK-DPU reads a $\mathcal{K}$ vector from its local Key Buffer and performs a vector dot-product operation. As such, while the $\mathcal{Q}$ vector is shared amongst the QK-DPU s, the $\mathcal{K}$ matrix is partitioned along its columns and is distributed across the Key Buffer s. Each QK-DPU performs the dot-product operations in a *bit-serial* mode, where the $\mathcal{K}$ elements are processed in bit-sequential manner and the $\mathcal{Q}$ elements are processed as a whole (e.g. 12 bit). Whenever each QK-DPU finishes the processing of all its $\mathcal{K}$ bits for unpruned *Score*s or early terminates the computation due to not meeting the layer pruning threshold based on the margin calculation described in Section 6.3.2, it proceeds with the execution of next $\mathcal{K}$ vector. If a QK-DPU detects a unpruned *Score*, it stores the *Score* value and its corresponding index on Score-FIFO and IDX-FIFO, respectively, to be processed by the back-end unit later. Once all the QK-DPU s finish processing all their $\mathcal{K}$ vectors, the QK-PU reads the next $\mathcal{Q}$ vector from Q-FIFO and starts its processing.

Bit-serial dot-product execution. Figure 6.5-(a) depicts the microarchitectural details of our Bit-Serial Dot-product Engine (BS-DPE). The BS-DPE is a collection of Multiply-ACcumulate (MAC) units and it performs a 12-bit$\times\mathcal{B}$-bit dot-product operation per cycle, where the $\mathcal{Q}$ vector is kept in a local register and $\mathcal{K}$s are read from the Key Buffer $\mathcal{B}$-bit at a time in a sequential mode. We chose $\mathcal{B} = 2$-bit as opposed to conventional bit-by-bit serial designs as the number of bits processed per cycle opens a unique trade-off space for the design of LEOPARD. Increasing

the bits leads to better power efficiency due to less frequent latching of intermediate results, however it may degrade the performance as it reduces the resolution of bit-level early termination. As such we perform a design space exploration (Figure 6.14 in Section 6.5.4) and chose 2-bit serial execution as it strikes the right balance between power efficiency and performance. The BS-DPE accumulates all the intermediate results in around 20 bits to keep required precision of the computations. The output of the last 2-bit$\times$12-bit MAC unit then goes to a shifter to scale the partial results according to the current $\mathcal{K}$ bit position and is accumulated and stored in a register that holds the (partial) results of *Score* computations.

Pruning detection via dynamic margin calculation. As discussed in Section 6.3.2 and Figure 6.3, to detect whether a current *Score* needs to be pruned and corresponding computations be terminated, QK-DPU dynamically calculates a *conservative upper-bound margin* ($\mathcal{M}$) and adds it with the current dot-product partial sum ($\mathcal{P}$) to compare it with the layer threshold ($\mathcal{T}h$). Figure 6.5-(b) and (c) show the details of hardware realization for margin calculation and thresholding logic, respectively. To calculate the margin according to Table in Figure 6.3, the margin calculation module first detects the $\mathcal{Q}$ and $\mathcal{K}$ pairs in the dot-product that yield positive product. To do so, during the processing of $\mathcal{K}$'s MSBs, the sign bits of $\mathcal{Q}$s and $\mathcal{K}$s are XORed. Only if the result is positive (XOR = 0), the absolute values of the corresponding $\mathcal{Q}$ are summed up to calculate the margin (e.g., resulting in $(9+5)$ in the Table of Figure 6.3). The summation result is stored in a Sum Register. Then, it is scaled by the fixed number, largest positive value (e.g. 0111...), which corresponds to $(2^{-1}+2^{-2}+2^{-3}+...)$ in Figure 6.3, storing $(9+5)(2^{-1}+2^{-2}+2^{-3}+...)$ in the margin register. The margin needs to be calculated dynamically for each bit position during bit-serial execution (such as $\mathcal{M}$ changing in each row of the Table in Figure 6.3). This is enabled by subtracting the shifted version of Sum Register value from the current margin in the margin register, e.g., $(9+5)(2^{-1}+2^{-2}+2^{-3}+...)-(9+5)(2^{-1}) = (9+5)(2^{-2}+2^{-3}+...)$ in the second row of the Table in Figure 6.3. This operation is iterated every bit position to generate the values in the subsequent rows of the Table in Figure 6.3. Note that, the margin calculation is a

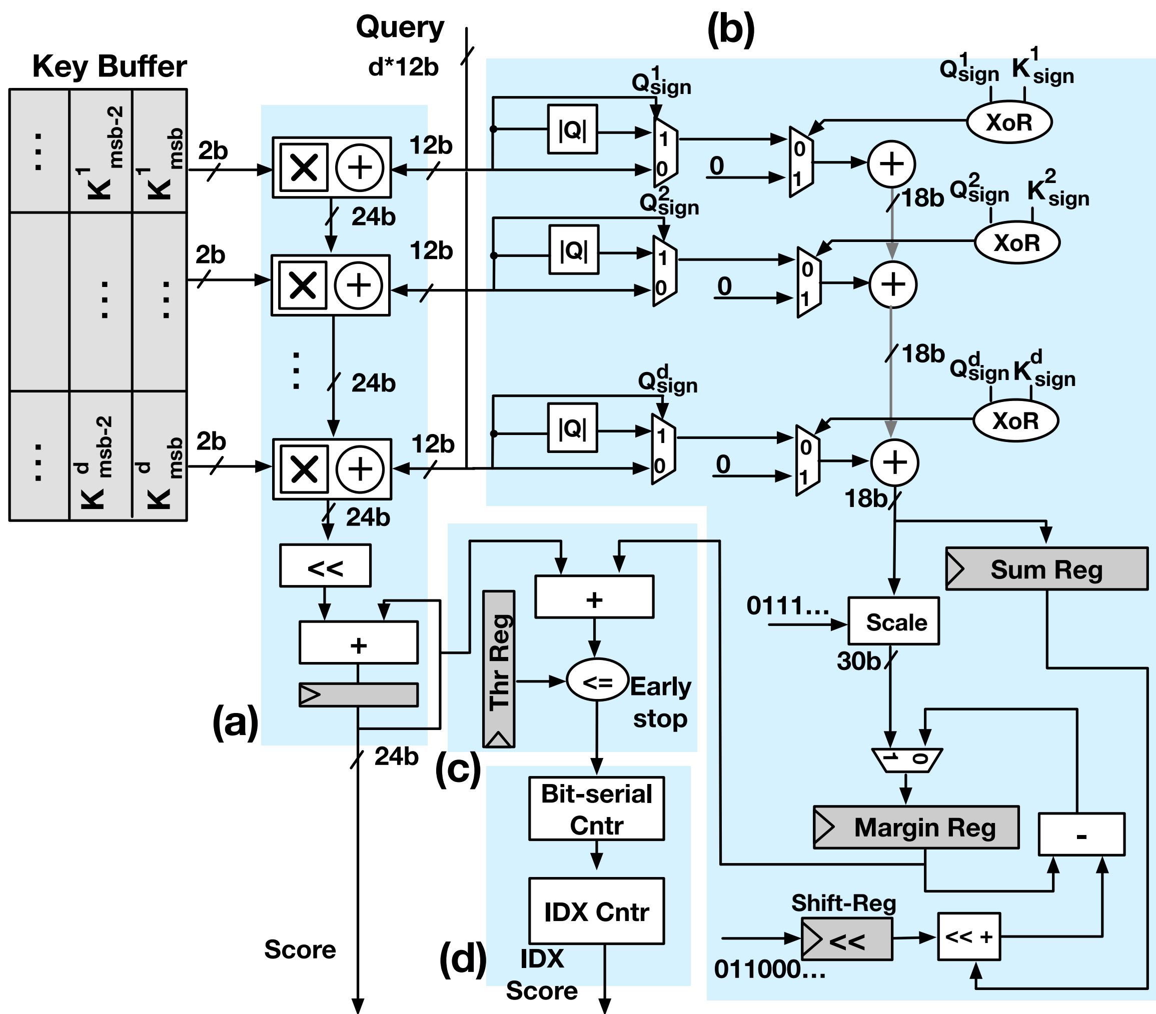


Figure 6.5. A QK-DPU comprising (a) bit-serial dot-product engine, (b) margin calculation logic, (c) thresholding module, and (d) score index counter.

scalar computation (mostly shift and subtraction), which is amortized over the $d = 64$ dimension vector processing, incurring virtually no overhead. After each cycle of the bit-serial operation, the thresholding module (Figure 6.5-(c)) adds the updated partial sum with the current margin and compares it with the layer threshold $\mathcal{T}h$ to determine the continuation of the dot-product or its termination for pruning of the current *Score*.

Final score index calculation. The QK-DPU calculates the indices of the unpruned *Score*s using a set of two counters, as shown in Figure 6.5-(d). First, Bit-serial Cntr increments with the number of bits processed by the QK-DPU and gets reset whenever it reaches its maximum (i.e. 6 (= 12bit/$\mathcal{B}$)) for processing all bits for unpruned *Score*s) or the Early stop flag is asserted. Second, the value of IDX Cntr shows the position of the current *Score* in the vector and increments whenever the Bit-serial Cntr gets reset, ending the computation of that *Score*. Finally, if the IDX Cntr increments and the Early stop flag is low, the QK-DPU pushes the content of this counter to IDX FIFO, because it means that the corresponding *Score* is not pruned and will be used for further processing in the V-PU.

6.4.3 Back-End Value Processing

As shown in Figure 6.4-(b), the LEOPARD tile's back-end stage, V-PU, consumes the unpruned *Score*s and executes the Softmax operation, followed by multiplication with $\mathcal{V}$ vectors and finally storing the results to an Output-FIFO. Whenever the Score-FIFO is not empty, the V-PU starts the Softmax operation (e^x and accumulation) to calculate the probabilities. We implemented the Softmax module of V-PU similarly to the Look-Up-Table (LUT)-based methodology in A^3 [103]. Whenever the output probability is produced, the V-PU uses the indices of the unpruned *Score*s to read the corresponding $\mathcal{V}$ vector. Finally, the $\mathcal{V}$ vector is weighted by the output of the Softmax module with a 1-D array of MAC units. The elements of $\mathcal{V}$ vector are distributed and the probabilities are shared across the MAC units, similar to a 1-D systolic array. With such design, the V-PU consumes the *Score*s sequentially to complete the weighted-sum of $\mathcal{V}$ vectors, and accumulates the partial results over multiple cycles while only accessing the

Table 6.1. Microarchitectural configurations of a LeOPArd tile.

Hardware modules	Configurations
QK-PU	6 / 8 QK-DPU (= N_{QK}), each 64 (= $\mathcal{D}$) tap 12×2 bit-serial
Key Buffer	48KB in total (= 8KB×6 / 6KB×8 banks), 128-bit port per bank
V-PU	Single 1-D 64 (= $\mathcal{D}$) way 16×16-bit MAC array
Value Buffer	64KB (= 8KB × 8 banks), 128-bit port per bank
Softmax	24-bit input, 16-bit output, LUT: 1 KB
Score and IDX FIFOs	24-bit × 512 depth for Score, 8-bit × 512 depth for IDX

unpruned $\mathcal{V}$ vectors. As such, it rightfully leverages the provided pruning by the front-end stage and eliminates the inconsequential computations.

6.5 Evaluation

6.5.1 Methodology

Workloads. To evaluate these models, we use five different datasets: (1) Facebook bAbI, which includes 20 different tasks [269] for MemN2N, (2) General Language Understanding Evaluation (GLUE) with nine different tasks [260] for BERT models, (3) Stanford Question Answering Dataset (SQUAD) [194] with a single task for BERT models and ALBERT-XX-L, (4) WikiText-2 [22] for GPT-2-L, and (5) CIFAR-10 [142] for ViT. The dimension (d) of $\mathcal{Q}$, $\mathcal{K}$, and $\mathcal{V}$ vectors for all the workloads is 64 except MemN2N with bAbI dataset, which is 20. The sequence length is 50 for MemN2N with bAbI whereas 512 and 384 for BERT and ALBERT-XX-L models with GLUE and SQUAD datasets, respectively. Finally, the sequence length for GPT-2 with WiKiText-2 is 1280.

Fine-tuning details. We use the baseline model checkpoints from HuggingFace [271] with PyTorch v1.10 and fine-tune the models on an Nvidia RTX 3090, except for GPT-2-Large, for which we use an Nvidia A100. For default task-level training, we use the Adam optimizer with default parameters and the learning rate of $[2,3] \times e^{-5}$ (same as baseline). To obtain the layer-specific threshold values, we perform an additional pruning-aware fine-tuning step for one to five more epochs to learn the optimal values while maintaining the baseline model accuracy.

For this step, we use the learning rate of $1e^{-2}$ for $\mathcal{T}h$ ($5e^{-6}$ for the other parameters), as training for the $\mathcal{T}h$ is generally slower and a higher learning rate facilitates convergence. To leverage faster fixed-point execution, we perform a final post-training quantization step with 12 bits for inputs in QK-PU hardware block and 16 bits for V-PU block similarly to [261].

Hardware design details. Table 6.1 lists the microarchitectural parameters of a single LEOPARD tile for two studied configurations: (1) A LEOPARD tile with six and (2) eight QK-DPU s that share a single 1-D MAC array in V-PU. The number of QK-DPU s is set such that the compute utilization for front-end and back-end units is balanced, while considering the pruning and bit-level early-termination rates across all the workloads. We synthesised and performed Placement-and-Route (P&R) for our designs with two tiles. The on-chip memory sizes for $\mathcal{K}$ and $\mathcal{V}$ are designed to store up to 512 sequences for a single head in a layer for both configurations.

Accelerator synthesis and simulations. We use Cadence Genus 19.1 [45] and Cadence Innovus 19.1 [46] to perform logic synthesis, floorplan, and P&R for the LEOPARD accelerator. We use TSMC 65 nm GP (General Purpose) standard cell library for the synthesis and layout generation of the digital logic blocks. These digital blocks are rigorously generated to meet the target frequency of 800MHz in consideration of all the CMOS corner variations and temperature conditions from -40° to 125°C. For the SRAM on-chip memory blocks, we use Memory Compiler with ARM High density 65 nm GP 6-transistor based single-port SRAM version r0p0 [34].

We also develop a simulator to obtain the total cycle counts and number of accesses to memories for both LEOPARD and baseline accelerators. The simulator incorporates the pruning rate and the bit-level early-termination statistics for each individual workload. Using these statistics, the simulator evaluates runtime and total energy consumption of the accelerators.

Comparison to baseline architecture. We compare LEOPARD to a conventional baseline design without any of our optimizations (e.g. runtime pruning and bit-level early compute termination). For a fair comparison, we use the same frequency, bitwidths for $\mathcal{Q} \times \mathcal{K}^T$ and $\times \mathcal{V}$,

and on-chip memory capacity for all the designs. The baseline design employs a single 12×12-bit QK-DPU as opposed to multiple 12×2-bit-serial ones, while both designs have the same back-end V-PU. As shown in Table 6.1, we evaluate LEOPARD under two design configurations. The first design with six QK-DPU s, dubbed Area-Efficient LEOPARD (AE-LEOPARD), almost perfectly matches the area of the baseline design (< 0.2% overhead) and provides an iso-area comparison setting. The second one with eight QK-DPU s, dubbed Highly-Parallel LEOPARD (HP-LEOPARD), provides an area 15% larger than baseline and delivers a better balance in the compute utilization of the front-end and back-end stages.

Comparison with A^3 and SpAtten. We also compare LEOPARD with two state-of-the-art attention accelerators, A^3 [103] and SpAtten [261], with support for runtime pruning. A^3 employs token pruning by comparing the Softmax output (probability) to a relative threshold, which is set using a user-defined parameter that adjusts the level of approximation. A^3 also employs a sorting mechanism to make the pruning decision after processing only a small number of large elements from the sorted $\mathcal{K}$ matrix in the order of magnitude. SpAtten performs cascaded head and token pruning by comparing the Softmax output with a user-defined threshold obtained empirically. There are no raw performance/energy results for individual workloads and simulation infrastructures of the accelerators. Therefore, we follow the comparison methodology of SpAtten [261], using throughput (GOPs / s), energy efficiency (GOPs / J), and area efficiency (GOPs / s / mm^2) metrics to provide the best comparisons. Both A^3 and SpAtten are implemented in 40 nm technology. To provide a fair comparison, we scale HP-LEOPARD from 65 nm to 40 nm based on both Dennard scaling (indicated with †) and measurement-based scaling rules [236] (indicated with ‡). We use a single tile with an area comparable to A^3 and SpAtten. Moreover, A^3 implements the $\mathcal{Q}\times\mathcal{K}^T$ using 9 bits as opposed to 12 bits in LEOPARD. As such, we scale the QK-PU of HP-LEOPARD from 12 bits to 9 bits to provide a head-to-head comparison with A^3.

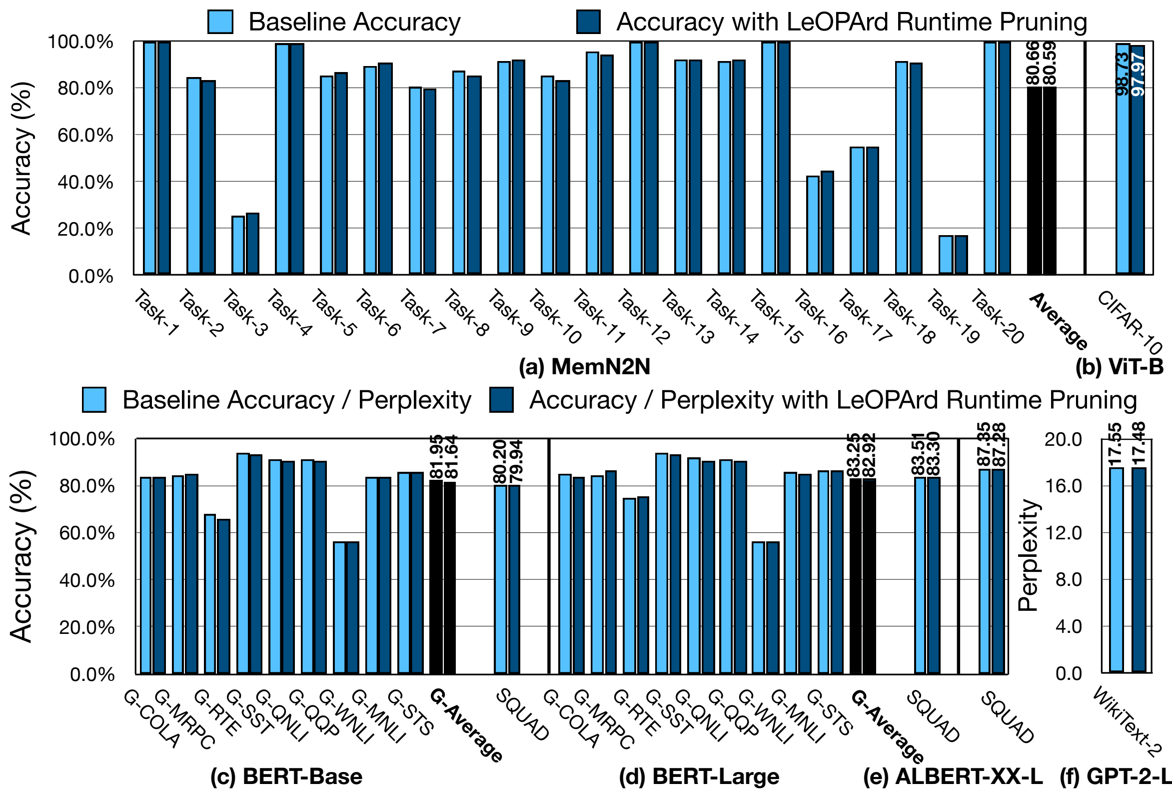


Figure 6.6. Accuracy before and after pruning-aware fine-tuning (prefix "G-": GLUE). We evaluate GPT-2 using perplexity, which favors a lower value.

6.5.2 Accuracy and Algorithmic Optimization

Impacts on model accuracy. Figure 6.6 compares the accuracies of the LEOPARD gradient-based on-the-fly pruning method and the baseline models in their vanilla implementation [271], across various tasks of evaluated workloads. On average, across all the evaluated tasks, LEOPARD runtime pruning degrades accuracy by only 0.07% for MemN2N with the bAbi dataset, 0.31% and 0.33% for BERT-B and BERT-L with the GLUE dataset, and 0.26% and 0.21% for BERT-B and BERT-L with the SQUAD dataset. For ALBERT-XX-L with the SQUAD dataset, the LEOPARD runtime pruning leads to only an 0.07% accuracy loss, whereas the degradation for ViT-B with the CIFAR-10 dataset is 0.76%.

In the GPT-2-L model, we use perplexity, which is the key metric for auto regressive language models. Note that perplexity is derived from the model loss, and thus lower perplexity is better. As shown in Figure 6.6-(f), LEOPARD runtime pruning results in a 0.07 decrease in perplexity. This is achievable because LEOPARD *learns* the optimal threshold values and co-adjusts them with the weight parameters simultaneously via gradient-based optimization. Figure 6.6 also illustrates that the LEOPARD pruning-aware fine-tuning pass evenly improves the accuracy for some of the benchmark tasks, with the maximum of 2.2%. However, this also degrades the accuracy for other tasks with the maximum of 2.6%. This accuracy fluctuations are unavoidable due to randomness in deep learning training, but overall the accuracy degradation, averaged across the evaluated benchmarks, converges adequately to a near-zero value ($\leq 0.2\%$). Performing the post-training quantization adds at most only 0.1%, for both the baseline and our pruning-aware fine-tuned models.

Runtime pruning rate analysis. Figure 6.7 shows the percentage of total $\mathcal{Q} \times \mathcal{K}^T$ *Score*s that are pruned away by our method using the learned threshold values across various benchmarks. In transformer software implementations, zeros are padded to maintain regular vector length despite the varying sequence length in each workload. The padded zeros are not counted for sparsity contribution in this work. On average, LEOPARD prunes 91.7% (max. 97.4%) of *Score*s

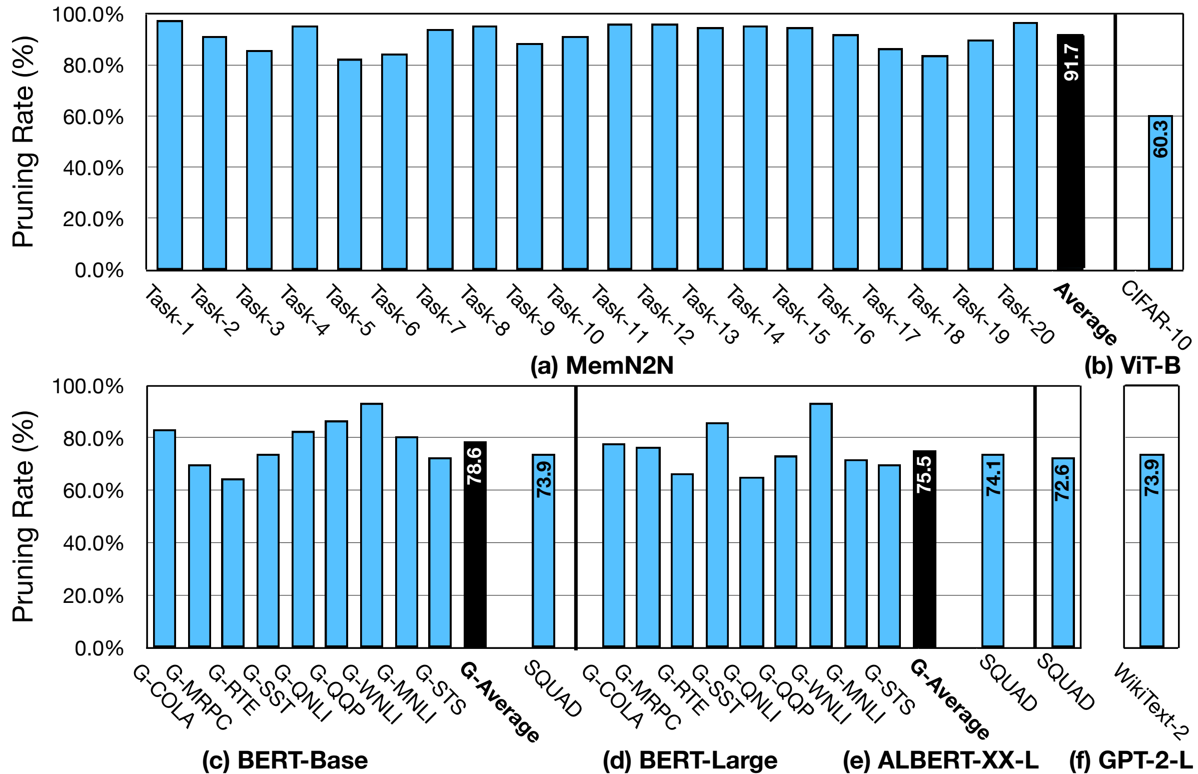


Figure 6.7. Runtime pruning rate with LeOPArd. (prefix "G-": GLUE)

across all the 20 tasks for the MeMN2N model with the bAbI dataset. LEOPARD achieves the average pruning rates of 78.6% (max. 93.2%) and 75.5% (max. 93.0%) for the BERT-B and BERT-L models with the GLUE dataset, while achieving 73.9% and 74.1% with the SQUAD dataset, respectively. Moreover, LEOPARD provides a 72.6% pruning rate for ALBERT-XX-L with the SQUAD dataset, 60.3% for ViT-B with the CIFAR-10 dataset, and 73.9% for GPT-2-L with the WikiText-2 dataset. As the results suggest, LEOPARD can significantly prune out the *Score*s across various tasks, with greater benefits to MeMN2N tasks compared to the BERT ones. We conjecture the lower pruning rates in BERT models are due to the higher probability of correlation between various tokens in the more complex language processing tasks compared to MemN2N.

As Figure 6.7 shows, in the case of ALBERT-XX-L with SQUAD, we see more pruning opportunities compared to BERT, presumably because of its larger model architecture with more redundant computations. Similar trend is observed for GPT-2-L. With regard to ViT-B,

we see lower pruning compared to NLP tasks, commensurate with prior studies [52]. This occurs because information is more local in images compared to texts, and therefore there is less redundancy in the attention layers for vision tasks.

Bit-level early-compute termination. Figure 6.8 depicts the proposed bit-level early compute termination feature and its relation with the achieved runtime pruning rates. The x-axis shows the number of bits processed sequentially, while the y-axis shows the cumulative achieved pruning rate averaged over all of the datasets' tasks. Intuitively, as more bits are processed during *Score* computations, the dynamic margin becomes smaller and thus the pruning rate increases. As shown, as the average number of processed bits increases, the cumulative pruning rate gradually plateaus, indicating saturation. In this scenario, the higher number of bits are only required for fully calculating unpruned *Score*s. We establish that the lower redundancy in model parameters of some transformer models, e.g. BERT-L / ViT-B, hinders higher runtime pruning. Because lower redundancy generally translates to a higher number of average bits calculations, it proportionally diminishes the potential gains from bit-wise early termination. Averaged over pruned *Score*s in bit-serial mode, MemN2N with the bAbi dataset requires 4.5 bits, while BERT-B and BERT-L require 8.3 and 8.0 bits with the GLUE dataset. With the SQUAD dataset, the average number of bits in BERT-B and BERT-L are 7.6 and 9.0 bits, whereas ALBERT-XX-L maintains 8.0 bits. The average number of bits in GPT-2-L and ViT attain 7.6 bits and 8.5 bits, respectively. This devised early-termination mechanism significantly reduces the computations of the $\mathcal{Q} \times \mathcal{K}^T$.

6.5.3 Accelerator Performance Results

Performance and energy comparison to baseline. Figure 6.9 shows the speedup improvements delivered by LEOPARD compared to the baseline design, across all the 43 studied tasks. In this comparison, we consider the total execution runtime for all attention layers of the models. On average across all tasks, AE-LEOPARD and HP-LEOPARD provide 1.9× and 2.4× speedup over the baseline, respectively. These improvements stem from both LEOPARD runtime pruning that reduces operations on the back-end unit (e.g., Softmax and $\times\mathcal{V}$) and bit-level early

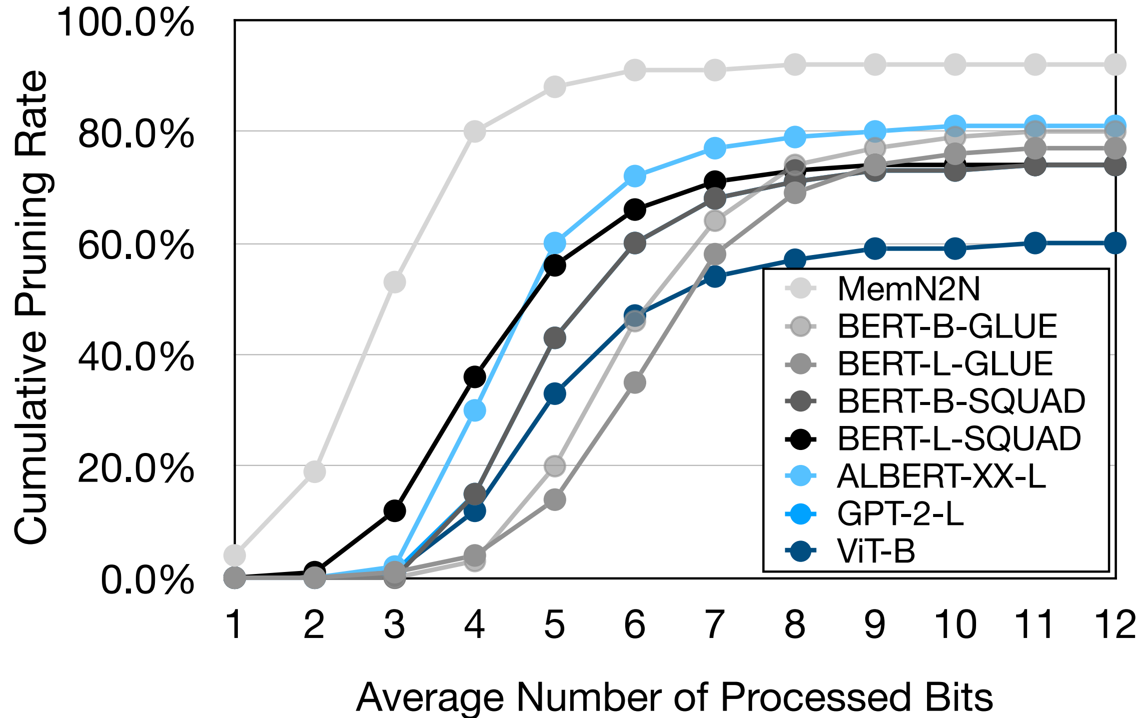


Figure 6.8. Cumulative pruning rate with respect to the number of bits processed during bit-serial early termination. Each line obtained by averaging across all the pruning rates per task.

compute termination that saves cycles on $\mathcal{Q} \times \mathcal{K}^T$ computations for pruned *Score*s. Across the workloads, LEOPARD delivers higher speedups for MemN2N compared to the other benchmarks. We attribute these improvements to the higher pruning rate and consequently more bit-level termination opportunities in this model's tasks. Among all the tasks, MemN2N-Task-1 enjoys the maximal speedup (3.8× for AE-LEOPARD and 5.1× for HP-LEOPARD) while ViT-B gains the minimal improvements (1.1× for both AE-LEOPARD and HP-LEOPARD). The benefits are more pronounced for HP-LEOPARD because it deploys more QK-DPU s, which both improves the performance of the front-end Q-PU unit, and delivers more inputs (*Score*s) to the back-end stage. The latter generally increases the back-end utilization.

Figure 6.10 compares the energy reduction (including compute and on-chip memory accesses) achieved by LEOPARD to the baseline. On average, LEOPARD reduces total energy consumption by a factor of 3.9× for AE-LEOPARD and 4.0× for HP-LEOPARD, across all the studied tasks. Similarly to the speedup comparisons, MemN2N enjoys a greater energy reduction

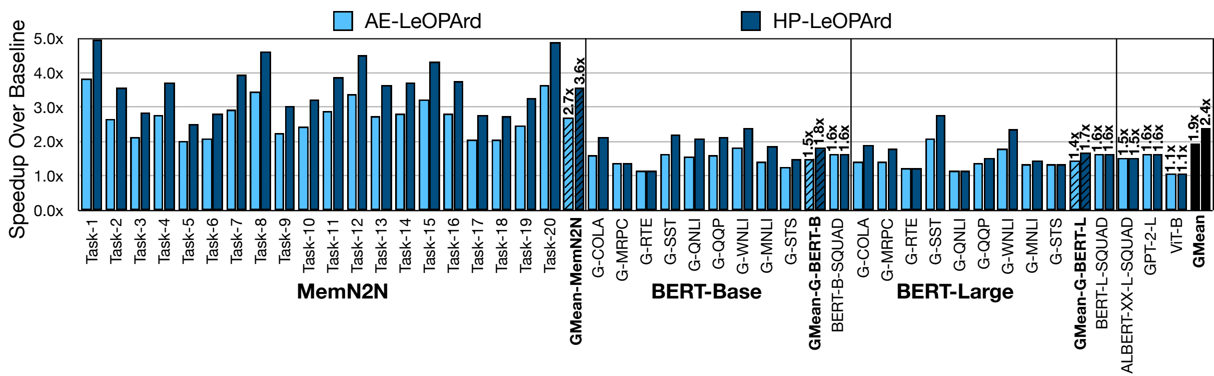


Figure 6.9. Speedup comparison to baseline design for AE-LeOPArd and HP-LeOPArd (prefix "G-": GLUE dataset).

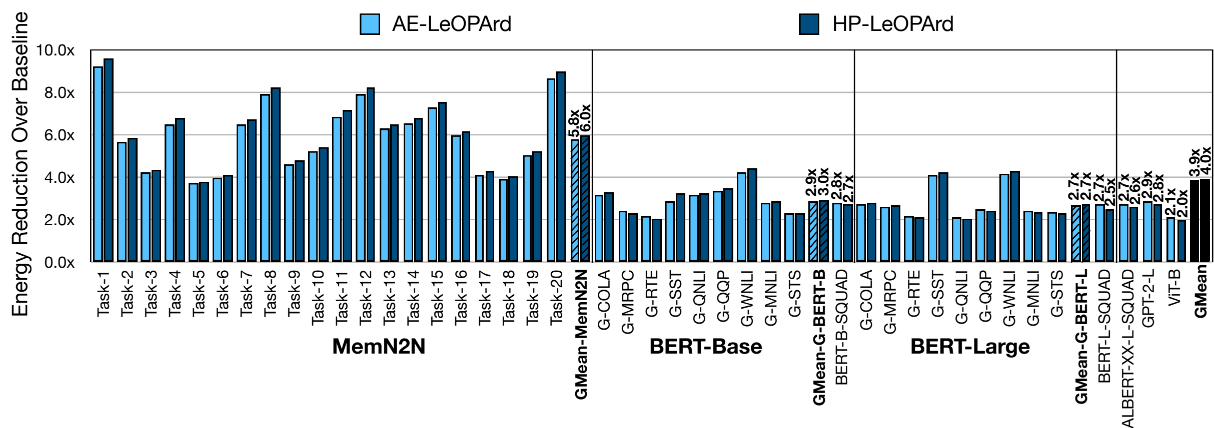


Figure 6.10. Total energy reduction for AE-LeOPArd and HP-LeOPArd compared to baseline (prefix "G-": GLUE dataset).

than the other benchmarks due to the higher pruning rate and therefore faster bit-level compute terminations. Across all tasks, the energy reduction is the greatest for MemN2N-Task-1 (9.2× for AE-LEOPARD and 9.6× for HP-LEOPARD) and ViT-B achieves the lowest savings (≈ 2.0× for AE-LEOPARD and HP-LEOPARD). The impact of LEOPARD on energy exceeds that on speedup, because runtime pruning and bit-level early termination reduce computation energy (contributing to both energy savings and speedup) and memory accesses (only contributing to energy savings). The energy reductions for both AE-LEOPARD and HP-LEOPARD are not substantially different. Because the additional QK-DPU s in HP-LEOPARD increase both power and performance, total energy consumption remains similar.

Analysis of energy savings breakdown. Figure 6.11 analyzes the breakdown of total energy consumption across five microarchitectural components: (1) $\mathcal{Q} \times \mathcal{K}^T$ computations, (2) $\mathcal{K}$ buffer memory access, (3) Softmax, (4) $\times\mathcal{V}$ computations, and (5) value buffer memory access. We report the average breakdown across all tasks for each workload. Additionally, Figure 6.11 illustrates the contribution of LEOPARD's two main optimizations: (1) runtime pruning and (2) early compute termination through bit-serial execution to the overall energy savings in AE-LEOPARD. We normalize the energy breakdowns to a baseline, which does not utilize any of the LEOPARD's optimizations. In the baseline, $\times\mathcal{V}$ computations and value buffer memory accesses proportionally consume the highest energy due to the lack of runtime pruning; ergo, higher average number of bits in $\mathcal{Q} \times \mathcal{K}^T$. Recall that the LEOPARD's back-end unit encloses Softmax, $\times\mathcal{V}$, and its associated buffer accesses. As the results show, this unit consumes more than 65% of the total energy in the baseline design. LEOPARD's runtime pruning enables skipping computations and memory accesses for inconsequential *Score*s during the back-end processing, delivering 1.7× (ViT-B) to 2.5× (MemN2N) energy savings. For these tasks, the bit-serial execution in LEOPARD along with its early termination brings further energy savings of 1.3× (ViT-B) to 2.3× (MemN2N) on top of runtime pruning. These additional benefits arise from avoiding the inconsequential bit computations in $\mathcal{Q} \times \mathcal{K}^T$ and their associated $\mathcal{K}$ buffer

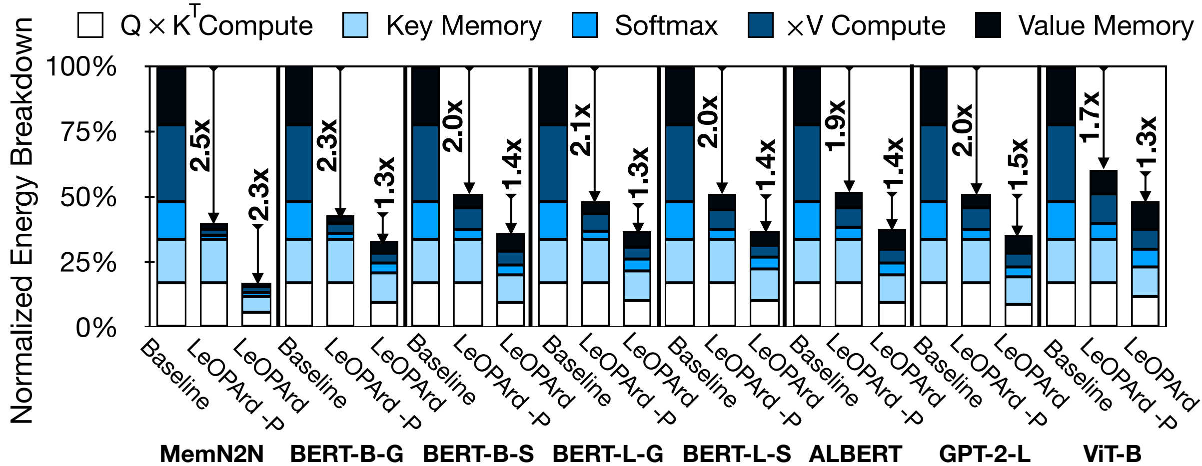


Figure 6.11. Normalized LeOPArd's average energy breakdown and the contribution of runtime pruning and bit-level early termination in energy saving (LeOPArd-P: with only pruning, and LeOPArd: pruning + bit-serial early termination) across one transformer head.

Table 6.2. LeOPArd performance comparison under different scenarios with prior work [103, 261].

Metric (unit)	A^3-Base	A^3-Conserv	SpAtten	HP-LeOPArd	HP-LeOPArd †	HP-LeOPArd ‡	HP-LeOPArd †*	HP-LeOPArd ‡*
Process (nm)	40	40	40	65	40	40	40	40
Area (mm^2)	2.08	2.08	1.55	3.47	1.31	1.31	1.05	1.05
Key Buffer (KB)	20	20	24	48	24	24	24	24
Value Buffer (KB)	20	20	24	64	24	24	24	24
($\mathcal{Q}$, $\mathcal{K}$)-bits	(9, 9)	(9, 9)	(12, 12)	(12, 12)	(12, 12)	(12, 12)	(9, 9)	(9, 9)
GOPs / s	259.0	518.0	728.4	574.1	932.8	1084.9	1143.9	1330.3
GOPs / J	2354.5	4709.1	772.9	519.3	2224.8	2028.8	3353.8	3058.4
GOPs / s / mm^2	124.5	249.0	470.0	165.5	710.4	826.1	1093.8	1272.1

† Dennard scaling trend applied to map on 40 nm process – ‡ Scaling rule from [236] applied to map on 40 nm process – *scaled to 9 bit $\mathcal{Q}$, $\mathcal{K}$

accesses.

Comparison with A^3 and SpAtten. Table 6.2 compares the characteristics and performance of HP-LeOPArd and its scaled versions with A^3 and SpAtten. Compared to SpAtten, HP-LeOPArd† (HP-LeOPArd‡) delivers 3× (2.6×) improvements in GOPs / J and 1.5× (1.7 ×) improvements in GOPs / s / mm^2, while both designs have virtually no model accuracy degradation. These benefits are attributed to the LeOPArd's higher pruning rate and to the bit-level early compute termination. For comparison with A^3, we evaluate HP-LeOPArd†* (HP-LeOPArd‡*), which are scaled to 40 nm and deploy 9-bit arithmetic for $\mathcal{Q} \times \mathcal{K}^T$. A^3-Conservative deploys heuristic approximation to minimize accuracy degradation on top of A^3-Base, which does not

use approximation. HP-LEOPARD$^{\dagger *}$ (HP-LEOPARD$^{\ddagger *}$) achieves 1.4× (1.3×) higher energy efficiency (in GOPs / J) and 8.8× (10.2×) area efficiency (in GOPs / s / mm^2) than A^3-base. HP-LEOPARD$^{\dagger *}$ (HP-LEOPARD$^{\ddagger *}$) also provides 4.4× (5.1×) improvements in terms of GOPs / s / mm^2 compared to A^3-Conservative. Although A^3-Conservative provides 29% and 35% higher energy efficiency compared to HP-LEOPARD$^{\dagger *}$ and HP-LEOPARD$^{\ddagger *}$, respectively, this comes at the cost of visible accuracy degradation, e.g., 1.0% for MemN2N and 1.3% for BERT-Base with the SQUAD dataset as reported in [103]. On the other hand, LEOPARD's carefully crafted gradient-based training balances pruning rate and model accuracy, providing accuracy degradation of only 0.06% and 0.26% for the aforementioned models and datasets without manual configurations for heuristic parameters.

LEOPARD accelerator layout area details. Figure 6.12(a) shows the layout of LEOPARD architecture, which occupies 2.3×2.8 mm^2, including two tiles. The layouts are generated by meeting the design rule check in a 65 nm process and targeting 65-75% physical density, commonly used for the routing convenience and tape-out yield. Figure 6.12-(b) reports the area breakdown, where QK-DPU takes the largest proportion as we employ N_{QK} QK-DPU in consideration of the high pruning rate. This leads to 56% area occupied by the front-end unit, which includes QK-DPU and $\mathcal{K}$ buffer. The on-chip memory for $\mathcal{K}$ and $\mathcal{V}$ occupies 34% of the layout area.

6.5.4 Architecture Design Space Exploration

QK-PU parallelism degree. As discussed in Section 6.4.1, the number of QK-DPU s (N_{QK}) within one QK-PU exhibits a trade-off space in designing the LEOPARD accelerator. To find the number of QK-DPU s that balances the utilization of front-end and back-end units, we sweep the N_{QK} from three to 12 in Figure 6.13 and report the V-PU utilization across the evaluated tasks. If utilization exceeds 100% (common when $N_{\text{QK}} = 12$), the back-end V-PU is over-subscribed due to the throughput mismatch between V-PU and QK-PU. This mismatch throttles the back-end V-PU and turns into the system bottleneck, frequently stalling the front-end. On the other hand,

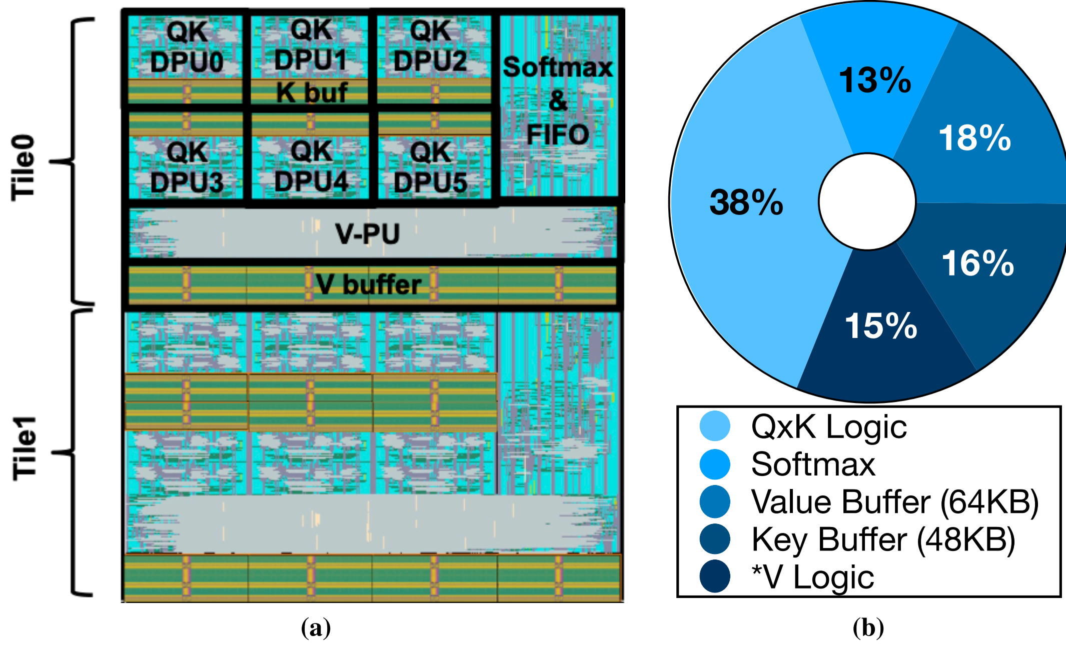


Figure 6.12. AE-LeOPArd: (a) layout (2.3×2.8 mm^2) and (b) area breakdown.

when $N_{QK} = 3$, the V-PU is chronically under-utilized due to a significant reduction in its number of computations, attributed to front-end runtime pruning mechanism. As marked by dark green diamonds, $N_{QK} = 8$ adequately balances the V-PU utilization and the number of front-end unit stalls. Thus, we favor this configuration for HP-LEOPARD. The second best configuration to balance front- and back-end utilization is $N_{QK} = 6$ (marked by light green diamonds). As such, we choose this configuration for AE-LEOPARD, which matches the baseline chip area usage.

Bit-serial processing granularity. Figure 6.14 illustrates the design space exploration for granularity of the bit-serial execution in QK-DPU ($\mathcal{B}$). This bit-level granularity creates a trade-off space, where decreasing the $\mathcal{B}$ stores intermediate results at the end of each bit processing cycle more frequently (escalating the energy). At the same time, increasing $\mathcal{B}$ curtails the performance of early compute termination due to lower resolution in stopping the computations. To find the optimal point, we sweep the $\mathcal{B}$ for values of 1, 2, 4, and 12 bits and measure the average consumed energy and its breakdown ($\mathcal{Q} \times \mathcal{K}^T$ logic and key buffer accesses) per one output *Score*. All the numbers are normalized to 12-bit processing that does not employ any bit-serial

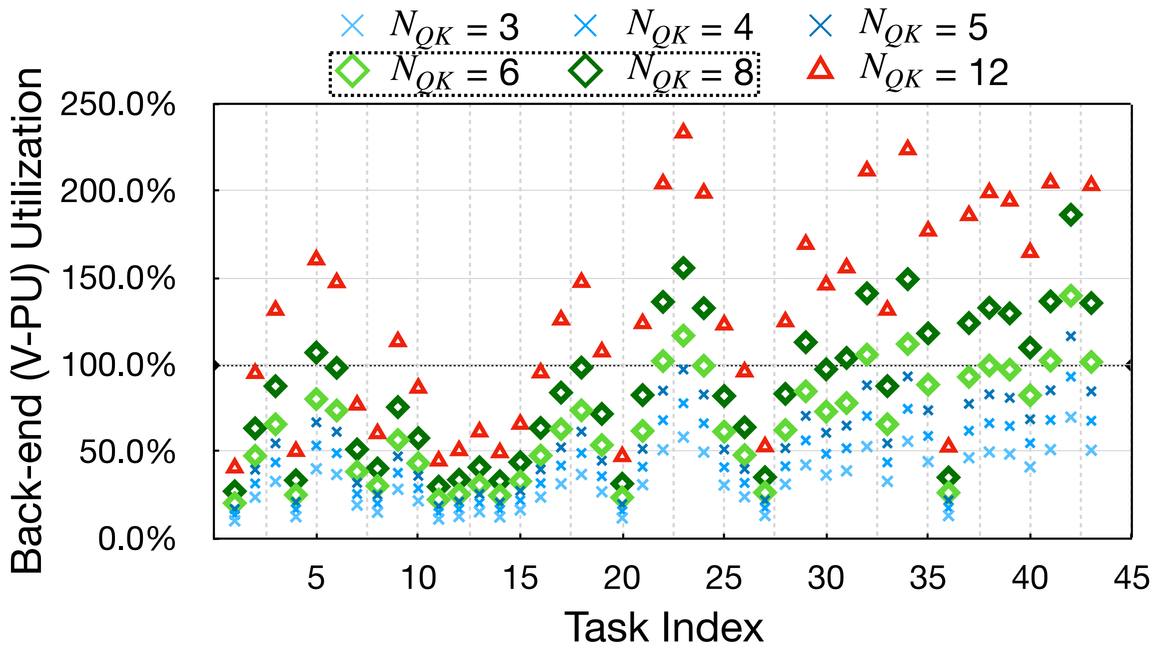


Figure 6.13. Back-end V-PU utilization over the QK-PU parallelism (N_{QK}). $N_{QK} = 6$ and $N_{QK} = 8$ form the favorable configurations in terms of back-end utilization in AE-LeOPArd and HP-LeOPArd, respectively.

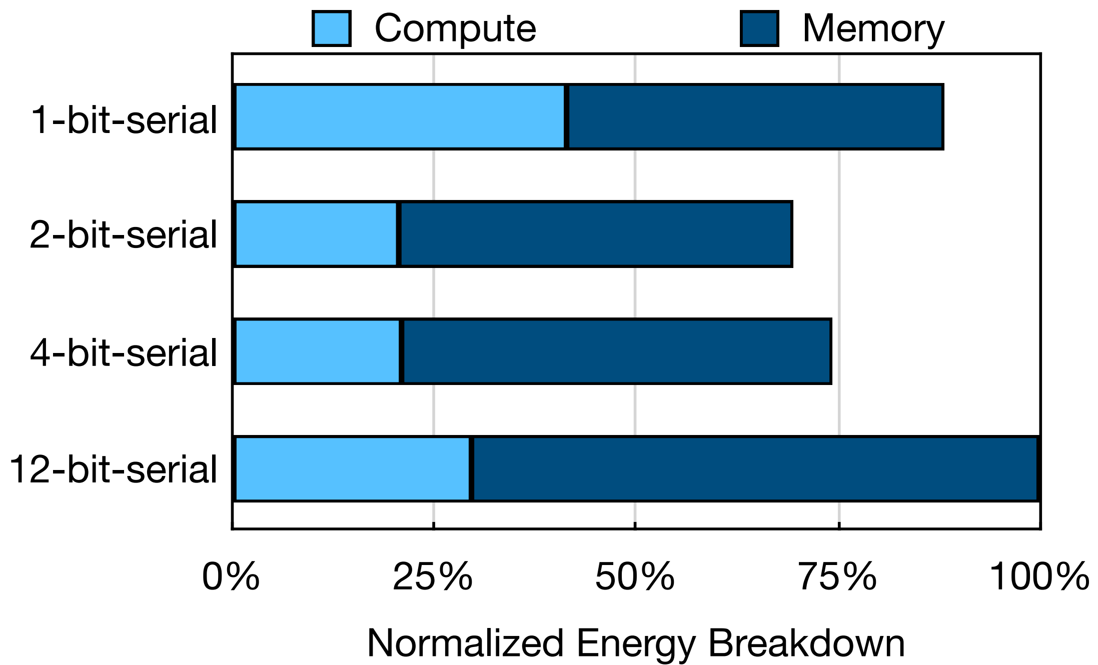


Figure 6.14. Design space exploration for the resolution of bit-serial execution with respect to normalized average QK-DPU energy per *Score*.

execution. Figure 6.14 depicts this analysis for MemN2N tasks (results for other models are similar) and reports the average across all tasks. As shown, 2-bit-serial execution strikes the right balance between energy consumption of the bit-serial computations and the resolution of bit-level early compute termination.

6.6 Related Work

In contrast with prior work, LEOPARD explores a distinct design space for accelerating attention models through gradient-based learned runtime pruning. This tight integration of pruning and training enables LEOPARD to reduce the computation cost with virtually zero accuracy degradation across a range of language and vision transformer models. Building on these algorithmic insights, we devise a bit-serial execution strategy that conservatively terminates the computations as early as possible. Below, we cover the most relevant work and position this research with respect to it.

Hardware-algorithm co-design for attention models. Several algorithmic optimizations co-designed with hardware acceleration were proposed for efficient execution of attention models [104, 261, 166, 243, 235, 103, 188, 287]. A^3 has proposed an *approximation* method with a hardware accelerator to prune out the ineffectual computations in attention. This method searches effective data during the $query \times key$ operation in addition to another approximation mechanism after score calculation. SpAtten [261] prunes the ineffectual input tokens and heads, in addition to progressive quantization during computations at runtime, to improve the performance and memory bandwidth. We provide a head-to-head comparison to these works in Section 6.5.3. ELSA [104] aims to address the costly candidate search process of A^3 and incorporates a user-defined "confidence-level" parameter to find the optimal thresholds from training statistics. EdgeBERT [243] leverages entropy-based early exiting technique to predict the minimal number of transformer layers that need to be executed, while the rest can be skipped. Other works aim to address the computational cost of self-attention via sparse matrix

operation [188, 166, 51], quantization [287], and Softmax approximation [235]. Moreover, none of these prior designs explored *bit-level* early compute termination.

Algorithmic optimizations for transformer acceleration. Another line of prior inquiry proposes only algorithmic optimizations to provide sparsity in computing attention models. Proposals in [204, 42, 284, 139, 192, 285, 60, 173, 266, 267] offer static sparsity in the attention layers to reduce its significant computational cost. Other work [297, 66, 68] provides dynamic sparsity based on the input samples, yet still requires full computation of the $\mathcal{Q}\times\mathcal{K}^T$. Our proposal fundamentally differs from this prior seminal work, because it formulates the problem of pruning threshold finding as a regularizer to methodically co-optimize with the weight parameters of the models, *without approximation*. Additionally, LEOPARD provides architectural support to stop the attention computations as early as possible during runtime.

Early compute-termination in DNNs. Prior work [223, 27, 149, 159] has proposed techniques to early terminate the computations of convolution layers by leveraging the zero production feature of ReLU for negative numbers. In contrast, this work focuses on early termination of a fundamentally different operator, attention in transformers, and provides unique mechanisms to enable that. Moreover, the prior works consider zero as a fixed threshold in their methods, but LEOPARD formulates the thresholds as a regularizer and *finds* layer-wise values through gradient descent optimization to preserve the accuracy of the models.

DNN acceleration. A large swath of work [214, 97, 98, 191, 96, 74, 211, 206, 219, 118, 94, 57, 216, 146, 147, 90, 222, 279, 203, 152, 280, 220, 80, 184, 28, 227, 92, 128, 106, 129, 196, 136, 215, 59, 295, 29, 161, 55, 221, 282, 234, 58] is dedicated to accelerating DNNs. Although inspiring, these designs do not deal with the challenges unique to the attention mechanisms of transformers, as opposed to this work.

6.7 Conclusion

Transformers through the self-attention mechanism have triggered an exciting new wave in machine learning, notably in Natural Language Processing (NLP). The self-attention mechanism computes pairwise correlations among all the words in a subtext. This task is both compute and memory intensive and has become one of the key challenges in realizing the full potential of attention models. One opportunity to slash the overheads of the self-attention mechanism is to limit the correlation computations to a few high score words and computationally prune the inconsequential scores at runtime through a thresholding mechanism. This work exclusively formulated the threshold finding as a gradient-based optimization problem. This formulation strikes a formal and analytical balance between model accuracy and computation reduction. To maximize the performance gains from thresholding, this research also devised a bit-serial architecture to enable an early-termination atop pruning with no repercussions to model accuracy. These techniques synergistically yield significant benefits both in terms of speedup and energy savings across various transformer-based models on a range of NLP and vision tasks. The application of the proposed mathematical formulation of identifying threshold values and its cohesive integration into the training loss is broad and can potentially be adopted across a wide range of compute reduction techniques.

6.8 Acknowledgement

Chapter 6 is a partial reprint of the material as it appears in: Z. Li, S. Ghodrati, A. Yazdanbakhsh, H. Esmaeilzadeh, M. Kang, "Accelerating Attention through Gradient-Based Learned Runtime Pruning." in *International Symposium on Computer Architecture (ISCA)*, 2022. The dissertation author, Zheng Li, and Amir Yazdanbakhsh were the primary investigators and contributed equally to this paper.

Bibliography

[1] Amazon alexa. https://developer.amazon.com/en-US/alexa/.

[2] Amazon case studies. https://aws.amazon.com/solutions/case-studies/.

[3] Amazon elastic inference. https://aws.amazon.com/machine-learning/elastic-inference/.

[4] Amazon sagemaker. https://aws.amazon.com/sagemaker/.

[5] Amazon sagemaker customers. https://aws.amazon.com/sagemaker/customers/.

[6] Apple a11-bionic. https://en.wikipedia.org/wiki/Apple_A11.

[7] Apple siri. https://www.apple.com/siri/.

[8] Azure machine learning. https://azure.microsoft.com/en-us/services/machine-learning/.

[9] Edge TPU. https://cloud.google.com/edge-tpu/.

[10] FreePDK45. https://www.eda.ncsu.edu/wiki/FreePDK45.

[11] Google assistant. https://assistant.google.com.

[12] Google cloud. https://cloud.google.com/products/ai/.

[13] Google cloud customers. https://cloud.google.com/customers/.

[14] Kubernetes. https://kubernetes.io.

[15] Nvdla. http://nvdla.org/index.html.

[16] Nvidia Jetson: The AI platform for autonomous machines. https://developer.nvidia.com/embedded/develop/hardware.

[17] Nvidia T4: Tensor core GPU for AI inference. https://www.nvidia.com/en-us/data-center/tesla-t4/.

[18] Nvidia tensor rt 5.1. https://developer.nvidia.com/tensorrt.

[19] Nvidia triton inference server. https://github.com/NVIDIA/triton-inference-server/.

[20] Zero-shot translation with google's multilingual neural machine translation system. https://ai.googleblog.com/2016/11/zero-shot-translation-with-googles.html.

[21] AI Winter. https://en.wikipedia.org/wiki/AI_winter, 2021. Accessed: 2021-11-08.

[22] The WikiText Long Term Dependency Language Modeling Dataset. https://blog.salesforceairesearch.com/the-wikitext-long-term-dependency-language-modeling-dataset/, 2021. Accessed: 2021-11-08.

[23] Turing Test. https://en.wikipedia.org/wiki/Turing_test, 2021. Accessed: 2021-11-08.

[24] gemmlowp: a small self-contained low-precision gemm library, 2022. https://github.com/google/gemmlowp.

[25] Byung Hoon Ahn, Jinwon Lee, Jamie Menjay Lin, Hsin-Pai Cheng, Jilei Hou, and Hadi Esmaeilzadeh. Ordering chaos: Memory-aware scheduling of irregularly wired neural networks for edge devices. In *MLSys*, 2020.

[26] Byung Hoon Ahn, Prannoy Pilligundla, and Hadi Esmaeilzadeh. Chameleon: Adaptive code optimization for expedited deep neural network compilation. In *ICLR*, 2020.

[27] Vahide Aklaghi, Amir Yazdanbakhsh, Kambiz Samadi, Hadi Esmaeilzadeh, and Rajesh K. Gupta. Snapea: Predictive early activation for reducing computation in deep convolutional neural networks. In *ISCA*, 2018.

[28] Jorge Albericio, Alberto Delmás, Patrick Judd, Sayeh Sharify, Gerard O'Leary, Roman Genov, and Andreas Moshovos. Bit-pragmatic deep neural network computing. In *MICRO*, 2017.

[29] Jorge Albericio, Patrick Judd, Tayler Hetherington, Tor Aamodt, Natalie Enright Jerger, and Andreas Moshovos. Cnvlutin: ineffectual-neuron-free deep neural network computing. In *ISCA*, 2016.

[30] Renzo Andri, Lukas Cavigelli, Davide Rossi, and Luca Benini. Yodann: An ultra-low power convolutional neural network accelerator based on binary weights. *arXiv*, 2016.

[31] Aayush Ankit, Izzat El Hajj, Sai Rahul Chalamalasetti, Geoffrey Ndu, Martin Foltin, R Stanley Williams, Paolo Faraboschi, Wen-mei W Hwu, John Paul Strachan, Kaushik Roy, and Dejan Milojicic. Puma: A programmable ultra-efficient memristor-based accelerator for machine learning inference. In *ASPLOS*, 2019.

[32] Aayush Ankit, Izzat El Hajj, Sai Rahul Chalamalasetti, Geoffrey Ndu, Martin Foltin, R Stanley Williams, Paolo Faraboschi, John Paul Strachan, Kaushik Roy, and Dejan S Milojicic. Puma: A programmable ultra-efficient memristor-based accelerator for machine learning inference. *arXiv preprint arXiv:1901.10351*, 2019.

[33] Andrew W. Appel. *Modern Compiler Implementation in ML: Basic Techniques*. Cambridge University Press, 1997.

[34] ARM. Artisan Memory Compilers. https://developer.arm.com/ip-products/physical-ip/embedded-memory, 2021. Accessed: 2021-11-08.

[35] Kambiz Azarian, Yash Bhalgat, Jinwon Lee, and Tijmen Blankevoort. Learned Threshold Pruning. 2020.

[36] Eunjin Baek, Dongup Kwon, and Jangwoo Kim. A multi-neural network acceleration architecture. *ISCA*, 2020.

[37] Daniel Bankman and Boris Murmann. Passive charge redistribution digital-to-analogue multiplier. *Electronics Letters*, 51(5):386–388, 2015.

[38] Daniel Bankman and Boris Murmann. An 8-bit, 16 input, 3.2 pj/op switched-capacitor dot product circuit in 28-nm fdsoi cmos. In *Solid-State Circuits Conference (A-SSCC), 2016 IEEE Asian*, pages 21–24. IEEE, 2016.

[39] Daniel Bankman, Lita Yang, Bert Moons, Marian Verhelst, and Boris Murmann. An always-on 3.8 μj/86% cifar-10 mixed-signal binary cnn processor with all memory on chip in 28nm cmos. In *ISSCC*, 2018.

[40] Michela Becchi, Kittisak Sajjapongse, Ian Graves, Adam Procter, Vignesh Ravi, and Srimat Chakradhar. A virtual memory based runtime to support multi-tenancy in clusters with gpus. In *HPDC*, 2012.

[41] Noah Beck, Sean White, Milam Paraschou, and Samuel Naffziger. 'zeppelin': An soc for multichip architectures. In *ISSCC*, 2018.

[42] Iz Beltagy, Matthew E Peters, and Arman Cohan. Longformer: The long-document transformer. *arXiv*, 2020.

[43] Robert W Brodersen, Paul R Gray, and David A Hodges. Mos switched-capacitor filters. *Proceedings of the IEEE*, 67(1):61–75, 1979.

[44] Fred N Buhler, Peter Brown, Jiabo Li, Thomas Chen, Zhengya Zhang, and Michael P Flynn. A 3.43 tops/w 48.9 pj/pixel 50.1 nj/classification 512 analog neuron sparse coding neural network with on-chip learning and classification in 40nm cmos. In *VLSI Circuits, 2017 Symposium on*, pages C30–C31. IEEE, 2017.

[45] Cadence. Genus Synthesis Solution. https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/synthesis/genus-synthesis-solution.html, 2021. Accessed: 2021-11-08.

[46] Cadence. Innovus Implementation System. https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/soc-implementation-and-floorplanning/innovus-implementation-system.html, 2021. Accessed: 2021-11-08.

[47] Guoyang Chen and Xipeng Shen. Free launch: optimizing gpu dynamic kernel launches through thread reuse. In *MICRO*, 2015.

[48] Lerong Chen, Jiawen Li, Yiran Chen, Qiuping Deng, Jiyuan Shen, Xiaoyao Liang, and Li Jiang. Accelerator-friendly neural-network training: learning variations and defects in rram crossbar. In *Proceedings of the Conference on Design, Automation & Test in Europe*, pages 19–24. European Design and Automation Association, 2017.

[49] Quan Chen, Hailong Yang, Minyi Guo, Ram Srivatsa Kannan, Jason Mars, and Lingjia Tang. Prophet: Precise qos prediction on non-preemptive accelerators to improve utilization in warehouse-scale computers. *ASPLOS*, 2017.

[50] Quan Chen, Hailong Yang, Jason Mars, and Lingjia Tang. Baymax: Qos awareness and increased utilization for non-preemptive accelerators in warehouse scale computers. *ASPLOS*, 2016.

[51] Tianlong Chen, Yu Cheng, Zhe Gan, Lu Yuan, Lei Zhang, and Zhangyang Wang. Chasing Sparsity in Vision Transformers: An End-to-End Exploration. 2021.

[52] Tianlong Chen, Yu Cheng, Zhe Gan, Lu Yuan, Lei Zhang, and Zhangyang Wang. Chasing sparsity in vision transformers:an end-to-end exploration. *arXiv*, 2021.

[53] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *OSDI*, 2018.

[54] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. Diannao: a small-footprint high-throughput accelerator for ubiquitous machine-learning. In *ASPLOS*, 2014.

[55] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *ISCA*, 2016.

[56] Yu-Hsin Chen, Tushar Krishna, Joel S Emer, and Vivienne Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *JSSC*, 2017.

[57] Yu-Hsin Chen, Tien-Ju Yang, Joel Emer, and Vivienne Sze. Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices. *JETCAS*, 2019.

[58] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. Dadiannao: A machine-learning supercomputer. In *MICRO*, 2014.

[59] Ping Chi, Shuangchen Li, Cong Xu, Tao Zhang, Jishen Zhao, Yongpan Liu, Yu Wang, and Yuan Xie. Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory. In *ISCA*, 2016.

[60] Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. Generating Long Sequences with Sparse Transformers. 2019.

[61] Jungwook Choi, Zhuo Wang, Swagath Venkataramani, Pierce I-Jen Chuang, Vijayalakshmi Srinivasan, and Kailash Gopalakrishnan. Pact: Parameterized clipping activation for quantized neural networks. *arXiv preprint arXiv:1805.06085*, 2018.

[62] Yujeong Choi and Minsoo Rhu. Prema: A predictive multi-task scheduling algorithm for preemptible neural processing units. *HPCA*, 2020.

[63] Krzysztof Choromanski, Valerii Likhosherstov, David Dohan, Xingyou Song, Andreea Gane, Tamas Sarlos, Peter Hawkins, Jared Davis, Afroz Mohiuddin, Lukasz Kaiser, David Belanger, Lucy Colwell, and Adrian Weller. Rethinking Attention with Performers. 2020.

[64] Eric Chung, Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Adrian Caulfield, Todd Massengil, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Christian Boehn, Oren Firestein, Alessandro Forin, Kang Su Gatlin, Mahdi Ghandi, Stephen Heil, Kyle Holohan, Tamas Juhasz, Ratna Kumar Kovvuri, Sitaram Lanka, Friedel van Megen, Dima Mukhortov, Prerak Patel, Steve Reinhardt, Adam Sapek, Raja Seera, Balaji Sridharan, Lisa Woods, Phillip Yi-Xiao, Ritchie Zhao, and Doug Burger. Accelerating persistent neural networks at datacenter scale. In *HotChips*, 2017.

[65] Hybrid Memory Cube Consortium. Hybrid memory cube specification 1.0. *Last Revision Jan*, 2013.

[66] Gonçalo M Correia, Vlad Niculae, and André FT Martins. Adaptively sparse transformers. *arXiv*, 2019.

[67] Jan Crols and Michel Steyaert. Switched-opamp: An approach to realize full cmos switched-capacitor circuits at very low power supply voltages. *IEEE Journal of Solid-State Circuits*, 29(8):936–942, 1994.

[68] Baiyun Cui, Yingming Li, Ming Chen, and Zhongfei Zhang. Fine-tune bert with sparse self-attention mechanism. In *EMNLP-IJCNLP*, 2019.

[69] Zihang Dai, Zhilin Yang, Yiming Yang, Jaime Carbonell, Quoc V Le, and Ruslan Salakhutdinov. Transformer-XL: Attentive Language Models Beyond a Fixed-length Context. 2019.

[70] Sudipto Das, Vivek R. Narasayya, Feng Li, and Manoj Syamala. Cpu sharing techniques for performance isolation in multi-tenant relational database-as-a-service. *Proc. VLDB Endow.*, 2013.

[71] Christina Delimitrou and Christos Kozyrakis. Quasar: resource-efficient and qos-aware cluster management. In *ASPLOS*, 2014.

[72] Christina Delimitrou, Daniel Sanchez, and Christos Kozyrakis. Tarcil: Reconciling Scheduling Speed and Quality in Large Shared Clusters. In *SoCC*, 2015.

[73] Alberto Delmas, Sayeh Sharify, Patrick Judd, and Andreas Moshovos. Tartan: Accelerating fully-connected and convolutional layers in deep learning networks by exploiting numerical precision variability. *arXiv*, 2017.

[74] Alberto Delmas Lascorz, Patrick Judd, Dylan Malone Stuart, Zissis Poulos, Mostafa Mahmoud, Sayeh Sharify, Milos Nikolic, Kevin Siu, and Andreas Moshovos. Bit-tactical: A software/hardware approach to exploiting value and bit sparsity in neural networks. In *ASPLOS*, 2019.

[75] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. Imagenet: A large-scale hierarchical image database. In *CVPR*, 2009.

[76] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc. Design of ion-implanted MOSFET's with very small physical dimensions. *JSSC*, 1974.

[77] ONNX Runtime developers. ONNX Runtime, 11 2018.

[78] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv*, 2018.

[79] Caiwen Ding, Siyu Liao, Yanzhi Wang, Zhe Li, Ning Liu, Youwei Zhuo, Chao Wang, Xuehai Qian, Yu Bai, Geng Yuan, Ma Xiaolong, Zhang Yipeng, Jian Tang, Qinru Qiu, Xue Lin, and Bo Yuan. Circnn: accelerating and compressing deep neural networks using block-circulant weight matrices. In *MICRO*, 2017.

[80] Charles Eckert, Xiaowei Wang, Jingcheng Wang, Arun Subramaniyan, Ravi Iyer, Dennis Sylvester, David Blaaauw, and Reetuparna Das. Neural cache: Bit-serial in-cache acceleration of deep neural networks. In *ISCA*, 2018.

[81] Yasuko Eckert, Nuwan Jayasena, and Gabriel H Loh. Thermal feasibility of die-stacked processing in memory. 2014.

[82] Ahmed T Elthakeb, Prannoy Pilligundla, FatemehSadat Mireshghallah, Amir Yazdanbakhsh, Sicun Gao, and Hadi Esmaeilzadeh. Releq: an automatic reinforcement learning approach for deep quantization of neural networks. *arXiv preprint arXiv:1811.01704*, 2018.

[83] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *ISCA*, 2011.

[84] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *ISCA*, 2011.

[85] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. Neural acceleration for general-purpose approximate programs. In *MICRO*, 2012.

[86] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. Neural acceleration for general-purpose approximate programs. *to apear in Commun. ACM*, 2013.

[87] Facebook AI Research. Caffe2. https://caffe2.ai/.

[88] Zhou Fang, Tong Yu, Ole J Mengshoel, and Rajesh K Gupta. Qos-aware scheduling of heterogeneous servers for inference in deep neural networks. In *CIKM*, 2017.

[89] John K Fiorenza, Todd Sepke, Peter Holloway, Charles G Sodini, and Hae-Seung Lee. Comparator-based switched-capacitor circuits for scaled cmos technologies. *IEEE Journal of Solid-State Circuits*, 41(12):2658–2668, 2006.

[90] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, Stephen Heil, Prerak Patel, Adam Sapek, Gabriel Weisz, Lisa Woods, Sitaram Lanka, Steven Reinhardt, Adrian Caulfield, Eric Chung, and Doug Burger. A configurable cloud-scale dnn processor for real-time ai. In *ISCA*, 2018.

[91] Trevor Gale, Erich Elsen, and Sara Hooker. The State of Sparsity in Deep Neural Networks. 2019.

[92] Mingyu Gao, Jing Pu, Xuan Yang, Mark Horowitz, and Christos Kozyrakis. Tetris: Scalable and efficient neural network acceleration with 3d memory. In *ASPLOS*, 2017.

[93] Mingyu Gao, Jing Pu, Xuan Yang, Mark Horowitz, and Christos Kozyrakis. Tetris: Scalable and efficient neural network acceleration with 3d memory. https://github.com/stanford-mast/nn_dataflow, 2017.

[94] Mingyu Gao, Xuan Yang, Jing Pu, Mark Horowitz, and Christos Kozyrakis. Tangram: Optimized coarse-grained dataflow for scalable nn accelerators. In *ASPLOS*, 2019.

[95] Hasan Genc, Seah Kim, Alon Amid, Ameer Haj-Ali, Vighnesh Iyer, Pranav Prakash, Jerry Zhao, Daniel Grubb, Harrison Liew, Howard Mao, Albert Ou, Colin Schmidt, Samuel Steffl, John Wright, Ion Stoica, Jonathan Ragan-Kelley, Krste Asanovic, Borivoje Nikolic, and Yakun Sophia Shao. Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration. In *DAC*, 2021.

[96] Soroush Ghodrati, Byung Hoon Ahn, Joon Kyung Kim, Sean Kinzer, Brahmendra Reddy Yatham, Navateja Alla, Hardik Sharma, Mohammad Alian, Eiman Ebrahimi, Nam Sung Kim, Cliff Young, and Hadi Esmaeilzadeh. Planaria: Dynamic Architecture Fission for Spatial Multi-Tenant Acceleration of Deep Neural Networks. In *MICRO*, 2020.

[97] Soroush Ghodrati, Hardik Sharma, Sean Kinzer, Amir Yazdanbakhsh, Jongse Park, Nam Sung Kim, Doug Burger, and Hadi Esmaeilzadeh. Mixed-signal charge-domain acceleration of deep neural networks through interleaved bit-partitioned arithmetic. In *PACT*, 2020.

[98] Soroush Ghodrati, Hardik Sharma, Cliff Young, Nam Sung Kim, and Hadi Esmaeilzadeh. Bit-parallel vector composability for neural acceleration. In *DAC*, 2020.

[99] Anousheh Gholami, Nariman Torkzaban, and John S Baras. On the importance of trust in next-generation networked cps systems: An ai perspective. *arXiv preprint arXiv:2104.07853*, 2021.

[100] Anousheh Gholami, Nariman Torkzaban, and John S Baras. Trusted decentralized federated learning. In *2022 IEEE 19th Annual Consumer Communications & Networking Conference (CCNC)*, pages 1–6. IEEE, 2022.

[101] Paul R Gray, Paul Hurst, Robert G Meyer, and Stephen Lewis. *Analysis and design of analog integrated circuits*. Wiley, 2001.

[102] Udit Gupta, Samuel Hsia, Vikram Saraph, Xiaodong Wang, Brandon Reagen, Gu-Yeon Wei, Hsien-Hsin S Lee, David Brooks, and Carole-Jean Wu. Deeprecsys: A system for optimizing end-to-end at-scale neural recommendation inference. *ISCA*, 2020.

[103] Tae Jun Ham, Sung Jun Jung, Seonghak Kim, Young H Oh, Yeonhong Park, Yoonho Song, Jung-Hun Park, Sanghee Lee, Kyoung Park, Jae W Lee, and Deog-Kyoon Jeong. A^ 3: Accelerating attention mechanisms in neural networks with approximation. In *HPCA*, 2020.

[104] Tae Jun Ham, Yejin Lee, Seong Hoon Seo, Soosung Kim, Hyunji Choi, Sung Jun Jung, and Jae W Lee. Elsa: Hardware-software co-design for efficient, lightweight self-attention mechanism in neural networks. In *ISCA*, 2021.

[105] Rehan Hameed, Wajahat Qadeer, Megan Wachs, Omid Azizi, Alex Solomatnikov, Benjamin C Lee, Stephen Richardson, Christos Kozyrakis, and Mark Horowitz. Understanding sources of inefficiency in general-purpose chips. In *ISCA*, 2010.

[106] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. Eie: efficient inference engine on compressed deep neural network. In *ISCA*, 2016.

[107] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. Toward dark silicon in servers. *IEEE Micro*, 2011.

[108] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. Toward dark silicon in servers. *IEEE Micro*, 31(4):6–15, July–Aug. 2011.

[109] Pieter Harpe. A 0.0013 mm2 10b 10ms/s sar adc with a 0.0048 mm2 42db-rejection passive fir filter. In *2018 IEEE Custom Integrated Circuits Conference, CICC 2018*. Institute of Electrical and Electronics Engineers Inc., 2018.

[110] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, 2016.

[111] Kartik Hegde, Rohit Agrawal, Yulun Yao, and Christopher W Fletcher. Morph: Flexible acceleration for 3d cnn-based video understanding. In *MICRO*, 2018.

[112] Kartik Hegde, Jiyong Yu, Rohit Agrawal, Mengjia Yan, Michael Pellauer, and Christopher W Fletcher. Ucnn: Exploiting computational reuse in deep neural networks via weight repetition. *arXiv*, 2018.

[113] John L Hennessy and David A Patterson. A new golden age for computer architecture. *CACM and Turing Lecture*, 2019.

[114] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: a platform for fine-grained resource sharing in the data center. In *NSDI*, 2011.

[115] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 1997.

[116] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv*, 2017.

[117] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Quantized neural networks: Training neural networks with low precision weights and activations. *arXiv*, 2016.

[118] Shehzeen Hussain, Mojan Javaheripi, Paarth Neekhara, Ryan Kastner, and Farinaz Koushanfar. Fastwave: Accelerating autoregressive convolutional neural networks on fpga. *arXiv*, 2020.

[119] Mohsen Imani, Saransh Gupta, Yeseong Kim, and Tajana Rosing. Floatpim: In-memory acceleration of deep neural network training with high precision. In *ISCA*, 2019.

[120] Mohammed Ismail and Terri Fiez. *Analog VLSI: signal and information processing*, volume 166. McGraw-Hill New York, 1994.

[121] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2704–2713, 2018.

[122] Joe Jeddeloh and Brent Keeth. Hybrid memory cube new dram architecture increases density and performance. In *VLSI Technology (VLSIT), 2012 Symposium on*, pages 87–88. IEEE, 2012.

[123] Natalie Enright Jerger, Dana Vantrease, and Mikko Lipasti. An evaluation of server consolidation workloads for multi-core designs. In *IISWC*, 2007.

[124] Houxiang Ji, Linghao Song, Li Jiang, Hai Halen Li, and Yiran Chen. Recom: An efficient resistive accelerator for compressed deep neural networks. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2018*, pages 237–240. IEEE, 2018.

[125] Yu Ji, Youyang Zhang, Xinfeng Xie, Shuangchen Li, Peiqi Wang, Xing Hu, Youhui Zhang, and Yuan Xie. Fpsa: A full system stack solution for reconfigurable reram-based nn accelerator architecture. *arXiv preprint arXiv:1901.09904*, 2019.

[126] Adwait Jog, Onur Kayiran, Tuba Kesten, Ashutosh Pattnaik, Evgeny Bolotin, Niladrish Chatterjee, Stephen W Keckler, Mahmut T Kandemir, and Chita R Das. Anatomy of gpu memory system for multi-application execution. In *MEMSYS*, 2015.

[127] Norman P Jouppi, Doe Hyun Yoon, Matthew Ashcraft, Mark Gottscho, Thomas B Jablin, George Kurian, James Laudon, Sheng Li, Peter Ma, Xiaoyu Ma, Tomas Norrie, Nishant Patil, Sushma Prasad, Cliff Young, Zongwei Zhou, and David Patterson. Ten lessons from three generations shaped google's tpuv4i: Industrial product. In *ISCA*, 2021.

[128] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Corriel, Mike Daley, Matt Dau, Dean Jeffrey, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon Mackean, Adriana Maggiore, Mair Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernikc, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Mathew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. In *ISCA*, 2017.

[129] Patrick Judd, Jorge Albericio, Tayler Hetherington, Tor M Aamodt, and Andreas Moshovos. Stripes: Bit-serial deep neural network computing. In *MICRO*, 2016.

[130] Ram Srivatsa Kannan, Lavanya Subramanian, Ashwin Raju, Jeongseob Ahn, Jason Mars, and Lingjia Tang. Grandslam: Guaranteeing slas for jobs in microservices execution frameworks. In *EuroSys*, 2019.

[131] Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, Nathan Pemberton, Emmanuel Amaro, Colin Schmidt, Aditya Chopra, Qijing Huang, Kyle Kovacs, Borivoje Nikolic, Katz Randy, Jonathan Bachhrachh, and Krste Asanovic. Firesim: Fpga-accelerated cycle-exact scale-out system simulation in the public cloud. In *ISCA*, 2018.

[132] Harshad Kasture and Daniel Sanchez. Ubik: Efficient Cache Sharing with Strict QoS for Latency-Critical Workloads. In *ASPLOS*, 2014.

[133] Angelos Katharopoulos, Apoorv Vyas, Nikolaos Pappas, and François Fleuret. Transformers are rnns: Fast autoregressive transformers with linear attention. In *ICML*, 2020.

[134] Jacob Devlin Ming-Wei Chang Kenton and Lee Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *NAACL-HLT*, 2019.

[135] Duckhwan Kim, Jaeha Kung, Sek Chai, Sudhakar Yalamanchili, and Saibal Mukhopadhyay. Neurocube: A programmable digital neuromorphic architecture with high-density 3d memory. In *ISCA*, 2016.

[136] Duckhwan Kim, Jaeha Kung, Sek Chai, Sudhakar Yalamanchili, and Saibal Mukhopadhyay. Neurocube: A programmable digital neuromorphic architecture with high-density 3d memory. In *ISCA*, 2016.

[137] Sehoon Kim, Amir Gholami, Zhewei Yao, Michael W Mahoney, and Kurt Keutzer. I-bert: Integer-only bert quantization. In *ICML*, 2021.

[138] Diederik P Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. 2014.

[139] Nikita Kitaev, Łukasz Kaiser, and Anselm Levskaya. Reformer: The efficient transformer. *arXiv*, 2020.

[140] John F. Kolen and Stefan C. Kremer. *Gradient Flow in Recurrent Nets: The Difficulty of Learning LongTerm Dependencies*. 2001.

[141] Alex Krizhevsky. One weird trick for parallelizing convolutional neural networks. *arXiv*, 2014.

[142] Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. *Computer Science Department, University of Toronto, Tech. Rep*, 2009.

[143] Anders Krogh and John A Hertz. A Simple Weight Decay can Improve Generalization. In *NIPS*, 1992.

[144] Solomon Kullback and Richard A Leibler. On Information and Sufficiency. *The annals of mathematical statistics*, 1951.

[145] HT Kung, Bradley McDanel, and Sai Qian Zhang. Packing sparse convolutional neural networks for efficient systolic array implementations: Column combining under joint optimization. In *ASPLOS*, 2019.

[146] Hyoukjun Kwon, Prasanth Chatarasi, Michael Pellauer, Angshuman Parashar, Vivek Sarkar, and Tushar Krishna. Understanding reuse, performance, and hardware cost of dnn dataflow: A data-centric approach. In *MICRO*, 2019.

[147] Hyoukjun Kwon, Ananda Samajdar, and Tushar Krishna. Maeri: Enabling flexible dataflow mapping over dnn accelerators via reconfigurable interconnects. *ASPLOS*, 2018.

[148] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. Albert: A lite bert for self-supervised learning of language representations. In *ICLR*, 2019.

[149] Dongwoo Lee, Sungbum Kang, and Kiyoung Choi. Compend: Computation pruning through early negative detection for relu in a deep neural network accelerator. In *ICS*, 2018.

[150] E. H. Lee and S. S. Wong. Analysis and design of a passive switched-capacitor matrix multiplier for approximate computing. *IEEE Journal of Solid-State Circuits*, 52(1):261–271, Jan 2017.

[151] Edward H Lee and S Simon Wong. Analysis and Design of a Passive Switched-Capacitor Matrix Multiplier for Approximate Computing. *IEEE Journal of Solid-State Circuits*, 52(1):261–271, 2017.

[152] Jinmook Lee, Changhyeon Kim, Sanghoon Kang, Dongjoo Shin, Sangyeob Kim, and Hoi-Jun Yoo. Unpu: A 50.6 tops/w unified deep neural network accelerator with 1b-to-16b fully-variable weight bit-precision. In *ISSCC*, 2018.

[153] Bing Li, Linghao Song, Fan Chen, Xuehai Qian, Yiran Chen, and Hai Helen Li. Reram-based accelerator for deep learning. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2018*, pages 815–820. IEEE, 2018.

[154] Fengfu Li, Bo Zhang, and Bin Liu. Ternary weight networks. *arXiv*, 2016.

[155] S. Li, K. Chen, J. H. Ahn, J. B. Brockman, and N. P. Jouppi. CACTI-P: Architecture-level Modeling for SRAM-based Structures with Advanced Leakage Reduction Techniques. In *ICCAD*, 2011.

[156] Sheng Li, Jung Ho Ahn, R.D. Strong, J.B. Brockman, D.M. Tullsen, and N.P. Jouppi. McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *MICRO*, 2009.

[157] Zheng Li, Soroush Ghodrati, Amir Yazdanbakhsh, Hadi Esmaeilzadeh, and Mingu Kang. Accelerating Attention through Gradient-Based Learned Runtime Pruning. In *ISCA*, 2022.

[158] Robert LiKamWa, Yunhui Hou, Julian Gao, Mia Polansky, and Lin Zhong. Redeye: analog convnet image sensor architecture for continuous mobile vision. In *ISCA*, 2016.

[159] Yingyan Lin, Charbel Sakr, Yongjune Kim, and Naresh Shanbhag. Predictivenet: An energy-efficient convolutional neural network via zero prediction. In *ISCAS*, 2017.

[160] Z. Lin, L. Nyland, and H. Zhou. Enabling efficient preemption for simt architectures with lightweight context switching. In *SC*, 2016.

[161] Shaoli Liu, Zidong Du, Jinhua Tao, Dong Han, Tao Luo, Yuan Xie, Yunji Chen, and Tianshi Chen. Cambricon: An instruction set architecture for neural networks. In *ISCA*, 2016.

[162] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C Berg. Ssd: Single shot multibox detector. In *ECCV*, 2016.

[163] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. *arXiv*, 2019.

[164] Yun Long, Taesik Na, and Saibal Mukhopadhyay. Reram-based processing-in-memory architecture for recurrent neural network acceleration. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, (99):1–14, 2018.

[165] Christos Louizos, Max Welling, and Diederik P Kingma. Learning Sparse Neural Networks through *L*_0 Regularization. 2017.

[166] Liqiang Lu, Yicheng Jin, Hangrui Bi, Zizhang Luo, Peng Li, Tao Wang, and Yun Liang. Sanger: A co-design framework for enabling sparse attention using reconfigurable architecture. In *MICRO*, 2021.

[167] Wenyan Lu, Guihai Yan, Jiajun Li, Shijun Gong, Yinhe Han, and Xiaowei Li. Flexflow: A flexible dataflow accelerator architecture for convolutional neural networks. In *HPCA*, 2017.

[168] Divya Mahajan, Jongse Park, Emmanuel Amaro, Hardik Sharma, Amir Yazdanbakhsh, Joon Kim, and Hadi Esmaeilzadeh. TABLA: A unified template-based framework for accelerating statistical machine learning. In *HPCA*, 2016.

[169] Mostafa Mahmoud, Kevin Siu, and Andreas Moshovos. Diffy: A déjà vu-free differential deep neural network accelerator. In *MICRO*, 2018.

[170] Mitchell P Marcus, Mary Ann Marcinkiewicz, and Beatrice Santorini. Building a large annotated corpus of english: The penn treebank. *Computational linguistics*, 1993.

[171] Jason Mars and Lingjia Tang. Whare-map: heterogeneity in" homogeneous" warehouse-scale computers. In *ISCA*, 2013.

[172] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *MICRO*, 2011.

[173] Paul Michel, Omer Levy, and Graham Neubig. Are Sixteen Heads Really Better than One? 2019.

[174] Facebook Research Microsoft. Onnx: an open format to represent deep learning models. http://onnx.ai/, 2017.

[175] Pascale Minet, Eric Renault, Ines Khoufi, and Selma Boumerdassi. Analyzing traces from a google data center. In *IWCMC*, 2018.

[176] Asit K. Mishra, Eriko Nurvitadhi, Jeffrey J. Cook, and Debbie Marr. WRPN: wide reduced-precision networks. *arXiv*, 2017.

[177] Daisuke Miyashita, Shouhei Kousai, Tomoya Suzuki, and Jun Deguchi. A neuromorphic chip optimized for deep learning and cmos technology with time-domain analog and digital mixed-signal processing. *IEEE Journal of Solid-State Circuits*, 52(10):2679–2689, 2017.

[178] Bert Moons, Roel Uytterhoeven, Wim Dehaene, and Marian Verhelst. DVAFS: Trading Computational Accuracy for Energy Through Dynamic-Voltage-Accuracy-Frequency-Scaling. In *DATE*, 2017.

[179] B. Murmann. *ADC Performance Survey 1997-2016*. murmann/adcsurvey.html, [Online]. Available.

[180] Kevin P Murphy. *Machine Learning: A Probabilistic Perspective*. MIT press, 2012.

[181] Wei Niu, Jiexiong Guan, Yanzhi Wang, Gagan Agrawal, and Bin Ren. Dnnfusion: accelerating deep neural networks execution with advanced operator fusion. In *PLDI*, 2021.

[182] NVIDIA. Nvidia turing architecture in-depth. https://developer.nvidia.com/blog/nvidia-turing-architecture-in-depth/, 2022.

[183] Mike O'Connor, Niladrish Chatterjee, Donghyuk Lee, John Wilson, Aditya Agrawal, Stephen W Keckler, and William J Dally. Fine-grained dram: energy-efficient dram for extreme bandwidth systems. In *MICRO*, pages 41–54. IEEE, 2017.

[184] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W Keckler, and William J Dally. SCNN: An Accelerator for Compressed-sparse Convolutional Neural Networks. In *ISCA*, 2017.

[185] Eunhyeok Park, Dongyoung Kim, and Sungjoo Yoo. Energy-Efficient Neural Network Accelerator Based on Outlier-Aware Low-Precision Computation. In *ISCA*, 2018.

[186] Jason Jong Kyu Park, Yongjun Park, and Scott Mahlke. Chimera: Collaborative preemption for multitasking on a shared gpu. *ASPLOS*, 2015.

[187] Jason Jong Kyu Park, Yongjun Park, and Scott Mahlke. Dynamic resource management for efficient utilization of multitasking gpus. *ASPLOS*, 2017.

[188] Junki Park, Hyunsung Yoon, Daehyun Ahn, Jungwook Choi, and Jae-Joon Kim. Optimus: Optimized matrix multiplication structure for transformer neural network accelerator. *MLSys*, 2020.

[189] Michael Pellauer, Yakun Sophia Shao, Jason Clemons, Neal Crago, Kartik Hegde, Rangharajan Venkatesan, Stephen W Keckler, Christopher W Fletcher, and Joel Emer. Buffets: An efficient and composable storage idiom for explicit decoupled data orchestration. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 137–151, 2019.

[190] Ximing Qiao, Xiong Cao, Huanrui Yang, Linghao Song, and Hai Li. Atomlayer: a universal reram-based cnn accelerator with atomic layer computation. In *Proceedings of the 55th Annual Design Automation Conference*, page 103. ACM, 2018.

[191] Eric Qin, Ananda Samajdar, Hyoukjun Kwon, Vineet Nadella, Sudarshan Srinivasan, Dipankar Das, Bharat Kaul, and Tushar Krishna. Sigma: A sparse and irregular gemm accelerator with flexible interconnects for dnn training. *HPCA*, 2020.

[192] Jiezhong Qiu, Hao Ma, Omer Levy, Scott Wen-tau Yih, Sinong Wang, and Jie Tang. Blockwise self-attention for long document understanding. *arXiv*, 2019.

[193] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. 2019.

[194] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. Squad: 100,000+ questions for machine comprehension of text. *arXiv*, 2016.

[195] Vignesh T. Ravi, Michela Becchi, Gagan Agrawal, and Srimat Chakradhar. Supporting gpu sharing in cloud environments with a transparent runtime consolidation framework. In *HPDC*, 2011.

[196] Brandon Reagen, Paul Whatmough, Robert Adolf, Saketh Rama, Hyunkwang Lee, Sae Kyu Lee, José Miguel Hernández-Lobato, Gu-Yeon Wei, and David Brooks. Minerva: Enabling low-power, highly-accurate deep neural network accelerators. In *ISCA*, 2016.

[197] Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole-Jean Wu, Brian Anderson, Maximilien Breughe, Mark Charlebois, William Chou, Ramesh Chukka, Cody Coleman, Sam Davis, Pan Deng, Greg Diamos, Jared Duke, Dave Fick, Gardner Scott, Itay Hubara, Sachin Idgunji, Tomas Jablin, Jeff Jiao, Tom John, Pankaj Kanwar, David Lee, Jeffery Liao, Anthon Lokhmotov, Francisco Massa, Peng Meng, Paulius Micikevicius, Colin Osborne, Gennady Pekhimenko, Arun Rajan, Dilip Sequeira, Ashish Sirasao, Fei Sun, Hanlin Tang, Michael Thomson, Frank Wei, Ephrem Wu, Lingjie Xu, Koichi Yamad, Bing Yu, George Yuan, Aaron Zhang, and Zhou Yuchen Zhang, Peizhao. Mlperf inference benchmark. *arxiv*, 2019.

[198] Joseph Redmon and Ali Farhadi. Yolo9000: better, faster, stronger. In *CVPR*, 2017.

[199] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. *arXiv*, 2018.

[200] Angad S Rekhi, Brian Zimmer, Nikola Nedovic, Ningxi Liu, Rangharajan Venkatesan, Miaorong Wang, Brucek Khailany, William J Dally, and C Thomas Gray. Analog/mixed-signal hardware error modeling for deep learning inference. In *Proceedings of the 56th Annual Design Automation Conference 2019*, page 81. ACM, 2019.

[201] Herbert Robbins and Sutton Monro. A Stochastic Approximation Method. *The annals of mathematical statistics*, 1951.

[202] Francisco Romero, Qian Li, Neeraja J Yadwadkar, and Christos Kozyrakis. INFaaS: Managed & model-less inference serving. *arXiv*, 2019.

[203] Bita Darvish Rouhani, Mohammad Samragh, Mojan Javaheripi, Tara Javidi, and Farinaz Koushanfar. Deepfense: Online accelerated defense against adversarial deep learning. In *ICCAD*, 2018.

[204] Aurko Roy, Mohammad Saffar, Ashish Vaswani, and David Grangier. Efficient content-based sparse attention with routing transformers. *TACL*, 2021.

[205] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning Internal Representations by Error Propagation. In *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*. MIT Press, 1986.

[206] Sungju Ryu, Hyungjun Kim, Wooseok Yi, and Jae-Joon Kim. Bitblade: Area and energy-efficient precision-scalable neural network accelerator with bitwise summation. In *DAC*, 2019.

[207] Sungju Ryu, Hyungjun Kim, Wooseok Yi, and Jae-Joon Kim. Bitblade: Area and energy-efficient precision-scalable neural network accelerator with bitwise summation. In *DAC 2019*, pages 1–6, 2019.

[208] Ananda Samajdar, Jan Moritz Joseph, Yuhao Zhu, Paul Whatmough, Matthew Mattina, and Tushar Krishna. A systematic methodology for characterizing scalability of dnn accelerators using scale-sim. In *ISPASS*, 2020.

[209] Ananda Samajdar, Yuhao Zhu, Paul Whatmough, Matthew Mattina, and Tushar Krishna. Scale-sim: Systolic cnn accelerator simulator. *arXiv*, 2018.

[210] Mohammad Samragh, Mojan Javaheripi, and Farinaz Koushanfar. Codex: Bit-flexible encoding for streaming-based fpga acceleration of dnns. *arXiv preprint arXiv:1901.05582*, 2019.

[211] Mohammad Samragh, Mojan Javaheripi, and Farinaz Koushanfar. Encodeep: Realizing bit-flexible encoding for deep neural networks. *TECS*, 2019.

[212] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *CVPR*, 2018.

[213] Dipanjan Sengupta, Anshuman Goswami, Karsten Schwan, and Krishna Pallavi. Scheduling multi-tenant cloud workloads on accelerator-based systems. In *SC*, 2014.

[214] Kiran Seshadri, Berkin Akin, James Laudon, Ravi Narayanaswami, and Amir Yazdanbakhsh. An Evaluation of Edge TPU Accelerators for Convolutional Neural Networks. In *IISWC*, 2022.

[215] Ali Shafiee, Anirban Nag, Naveen Muralimanohar, Rajeev Balasubramonian, John Paul Strachan, Miao Hu, R Stanley Williams, and Vivek Srikumar. Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars. In *ISCA*, 2016.

[216] Yakun Sophia Shao, Jason Clemons, Rangharajan Venkatesan, Brian Zimmer, Matthew Fojtik, Nan Jiang, Ben Keller, Alicia Klinefelter, Nathaniel Pinckney, Priyanka Raina, Stephen Tell, Yanqing Zhang, William Dally, Joel Emer, Thomas Gray, Brucek Khailany, and Stephen Keckler. Simba: Scaling deep-learning inference with multi-chip-module-based architecture. In *MICRO*, 2019.

[217] Sayeh Sharify, Alberto Delmas Lascorz, Patrick Judd, and Andreas Moshovos. Loom: Exploiting weight and activation precisions to accelerate convolutional neural networks. *arXiv*, 2017.

[218] Sayeh Sharify, Alberto Delmas Lascorz, Mostafa Mahmoud, Milos Nikolic, Kevin Siu, Dylan Malone Stuart, Zissis Poulos, and Andreas Moshovos. Laconic deep learning inference acceleration. In *ISCA*, 2019.

[219] Sayeh Sharify, Alberto Delmas Lascorz, Mostafa Mahmoud, Milos Nikolic, Kevin Siu, Dylan Malone Stuart, Zissis Poulos, and Andreas Moshovos. Laconic deep learning inference acceleration. In *ISCA*, 2019.

[220] Sayeh Sharify, Alberto Delmas Lascorz, Kevin Siu, Patrick Judd, and Andreas Moshovos. Loom: Exploiting weight and activation precisions to accelerate convolutional neural networks. In *DAC*, 2018.

[221] Hardik Sharma, Jongse Park, Divya Mahajan, Emmanuel Amaro, Joon Kim, Chenkai Shao, Asit Misra, and Hadi Esmaeilzadeh. From high-level deep neural models to fpgas. In *MICRO*, 2016.

[222] Hardik Sharma, Jongse Park, Naveen Suda, Liangzhen Lai, Benson Chau, Vikas Chandra, and Hadi Esmaeilzadeh. Bit fusion: Bit-level dynamically composable architecture for accelerating deep neural networks. *ISCA*, 2018.

[223] Gil Shomron, Ron Banner, Moran Shkolnik, and Uri Weiser. Thanks for nothing: Predicting zero-valued activations with lightweight convolutional neural networks. In *ECCV*, 2020.

[224] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv*, 2014.

[225] Tripti Singhal. Maximizing gpu utilization for datacenter inference with nvidia tensorrt inference server. 2019.

[226] James E. Smith and Andrew R. Pleszkun. Implementing precise interrupts in pipelined processors. *TC*, 1988.

[227] Linghao Song, Xuehai Qian, Hai Li, and Yiran Chen. Pipelayer: A pipelined reram-based accelerator for deep learning. In *HPCA*, 2017.

[228] Linghao Song, Xuehai Qian, Hai Li, and Yiran Chen. Pipelayer: A pipelined reram-based accelerator for deep learning. In *HPCA*, 2017.

[229] Mingcong Song, Jiechen Zhao, Yang Hu, Jiaqi Zhang, and Tao Li. Prediction based execution on deep neural networks. In *ISCA*, 2018.

[230] Suraj Srinivas, Akshayvarun Subramanya, and R Venkatesh Babu. Training Sparse Neural Networks. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition Workshops*, 2017.

[231] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *The journal of machine learning research*, 2014.

[232] Nitish Srivastava, Hanchen Jin, Jie Liu, David Albonesi, and Zhiru Zhang. Matraptor: A sparse-sparse matrix multiplication accelerator based on row-wise product. In *MICRO*, 2020.

[233] Prakalp Srivastava, Mingu Kang, Sujan K Gonugondla, Sungmin Lim, Jungwook Choi, Vikram Adve, Nam Sung Kim, and Naresh Shanbhag. Promise: An end-to-end design of a programmable mixed-signal accelerator for machine-learning algorithms. In *ISCA*, 2018.

[234] Renée St. Amant, Amir Yazdanbakhsh, Jongse Park, Bradley Thwaites, Hadi Esmaeilzadeh, Arjang Hassibi, Luis Ceze, and Doug Burger. General-purpose code acceleration with limited-precision analog computation. In *ISCA*, 2014.

[235] Jacob R Stevens, Rangharajan Venkatesan, Steve Dai, Brucek Khailany, and Anand Raghunathan. Softermax: Hardware/software co-design of an efficient softmax for transformers. *arXiv*, 2021.

[236] Aaron Stillmaker and Bevan Baas. Scaling Equations for the Accurate Prediction of CMOS Device Performance from 180 nm to 7 nm. *Integration*, 2017.

[237] Emma Strubell, Ananya Ganesh, and Andrew McCallum. Energy and policy considerations for deep learning in nlp. *arXiv preprint arXiv:1906.02243*, 2019.

[238] Sainbayar Sukhbaatar, arthur szlam, Jason Weston, and Rob Fergus. End-to-end memory networks. In *NeurIPS*, 2015.

[239] H Ekin Sumbul, Tony F Wu, Yuecheng Li, Syed Shakib Sarwar, William Koven, Eli Murphy-Trotzky, Xingxing Cai, Elnaz Ansari, Daniel H Morris, Huichu Liu, Kim Doyun, and Edith Beigne. System-Level Design and Integration of a Prototype AR/VR Hardware Featuring a Custom Low-Power DNN Accelerator Chip in 7nm Technology for Codec Avatars. In *CICC*, 2022.

[240] Lalith Suresh, Peter Bodik, Ishai Menache, Marco Canini, and Florin Ciucu. Distributed resource management across process boundaries. In *SoCC*, 2017.

[241] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *CVPR*, 2015.

[242] Abdulaziz Tabbakh, Murali Annavaram, and Xuehai Qian. Power efficient sharing-aware gpu data management. In *IPDPS*, 2017.

[243] Thierry Tambe, Coleman Hooper, Lillian Pentecost, Tianyu Jia, En-Yu Yang, Marco Donato, Victor Sanh, Paul Whatmough, Alexander M Rush, David Brooks, and Gu-Yeon Wei. Edgebert: Sentence-level energy optimizations for latency-aware multi-task nlp inference. In *MICRO*, 2021.

[244] Mingxing Tan and Quoc V Le. Efficientnet: Rethinking model scaling for convolutional neural networks. *ICML*, 2019.

[245] Ivan Tanasic, Isaac Gelado, Javier Cabezas, Alex Ramirez, Nacho Navarro, and Mateo Valero. Enabling preemptive multiprogramming on gpus. In *ISCA*, 2014.

[246] Lingjia Tang, Jason Mars, and Mary Lou Soffa. Compiling for niceness: Mitigating contention for qos in warehouse scale computers. In *CGO*, 2012.

[247] Tesla. Dojo chip. https://www.tesla.com/AI, 2022.

[248] Prashanth Thinakaran, Jashwant Raj Gunasekaran, Bikash Sharma, Mahmut Taylan Kandemir, and Chita R Das. Kube-knots: Resource harvesting through dynamic container orchestration in gpu-based datacenters. In *CLUSTER*, 2019.

[249] Prashanth Thinakaran, Jashwant Raj, Bikash Sharma, Mahmut T Kandemir, and Chita R Das. The curious case of container orchestration and scheduling in gpu-based datacenters. In *SoCC*, 2018.

[250] Nariman Torkzaban and John S Baras. Trust-aware service function chain embedding: A path-based approach. In *2020 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, pages 31–36. IEEE, 2020.

[251] Nariman Torkzaban, Chrysa Papagianni, and John S Baras. Trust-aware service chain embedding. In *2019 Sixth International Conference on Software Defined Systems (SDS)*, pages 242–247. IEEE, 2019.

[252] Vaibhav Tripathi and Boris Murmann. Mismatch characterization of small metal fringe capacitors. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 61(8):2236–2242, 2014.

[253] YP Tsividis and D Anastassiou. Switched-capacitor neural networks. *Electronics Letters*, 23(18):958–959, 1987.

[254] Y. Ukidave, X. Li, and D. Kaeli. Mystic: Predictive scheduling for gpu based cloud servers using machine learning. In *IPDPS*, 2016.

[255] Balajee Vamanan, Hamza Bin Sohail, Jahangir Hasan, and TN Vijaykumar. Timetrader: Exploiting latency tail to save datacenter energy for online search. In *MICRO*, 2015.

[256] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is All You Need. In *NeurIPS*, 2017.

[257] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph Attention Networks. 2017.

[258] Ganesh Venkatesh, Jack Sampson, Nathan Goulding, Saturnino Garcia, Vladyslav Bryksin, Jose Lugo-Martinez, Steven Swanson, and Michael Bedford Taylor. Conservation cores: Reducing the energy of mature computations. In *ASPLOS*, 2010.

[259] Nandita Vijaykumar, Kevin Hsieh, Gennady Pekhimenko, Samira Khan, Ashish Shrestha, Saugata Ghose, Adwait Jog, Phillip B Gibbons, and Onur Mutlu. Zorua: A holistic approach to resource virtualization in gpus. In *MICRO*, 2016.

[260] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R Bowman. Glue: A multi-task benchmark and analysis platform for natural language understanding. *arXiv*, 2018.

[261] Hanrui Wang, Zhekai Zhang, and Song Han. Spatten: Efficient sparse attention architecture with cascade token and head pruning. In *HPCA*, 2021.

[262] Sinong Wang, Belinda Z Li, Madian Khabsa, Han Fang, and Hao Ma. Linformer: Self-attention with linear complexity. *arXiv*, 2020.

[263] Wei Wang, Tanima Dey, Jason Mars, Lingjia Tang, Jack W Davidson, and Mary Lou Soffa. Performance analysis of thread mappings with a holistic view of the hardware resources. In *ISPASS*, 2012.

[264] Yuzhao Wang, Lele Li, You Wu, Junqing Yu, Zhibin Yu, and Xuehai Qian. Tpshare: a time-space sharing scheduling abstraction for shared cloud via vertical labels. In *ISCA*, 2019.

[265] Z. Wang, J. Yang, R. Melhem, B. Childers, Y. Zhang, and M. Guo. Simultaneous multikernel gpu: Multi-tasking throughput processors via fine-grained sharing. In *HPCA*, 2016.

[266] Ziheng Wang, Jeremy Wohlwend, and Tao Lei. Structured Pruning of Large Language Models. 2019.

[267] Wei Wen, Yuxiong He, Samyam Rajbhandari, Minjia Zhang, Wenhan Wang, Fang Liu, Bin Hu, Yiran Chen, and Hai Li. Learning Intrinsic Sparse Structures within Long Short-Term Memory. 2017.

[268] David Wentzlaff, Charles Gruenwald III, Nathan Beckmann, Kevin Modzelewski, Adam Belay, Lamia Youseff, Jason Miller, and Anant Agarwal. An operating system for multicore and clouds: Mechanisms and implementation. In *SoCC*, 2010.

[269] Jason Weston, Antoine Bordes, Sumit Chopra, Alexander M Rush, Bart van Merriënboer, Armand Joulin, and Tomas Mikolov. Towards ai-complete question answering: A set of prerequisite toy tasks. *arXiv*, 2015.

[270] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

[271] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander Rush. HuggingFace's Transformers: State-of-the-Art Natural Language Processing. 2019.

[272] Bo Wu, Guoyang Chen, Dong Li, Xipeng Shen, and Jeffrey Vetter. Enabling and exploiting flexible task assignment on gpu through sm-centric program transformations. In *ICS*, 2015.

[273] Bo Wu, Xu Liu, Xiaobo Zhou, and Changjun Jiang. Flep: Enabling flexible and efficient preemption on gpus. In *ASPLOS*, 2017.

[274] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Lukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Dean Jeffrey. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv*, 2016.

[275] Rui Xu, Sheng Ma, Yaohua Wang, Yang Guo, Dongsheng Li, and Yuran Qiao. Heterogeneous Systolic Array Architecture for Compact CNNs Hardware Accelerators. *IEEE Transactions on Parallel and Distributed Systems*, 2021.

[276] Dingqing Yang, Amin Ghasemazar, Xiaowei Ren, Maximilian Golub, Guy Lemieux, and Mieszko Lis. Procrustes: a Dataflow and Accelerator for Sparse Deep Neural Network Training. In *MICRO*, 2020.

[277] Tzu-Hsien Yang, Hsiang-Yun Cheng, Chia-Lin Yang, I Tseng, Han-Wen Hu, Hung-Sheng Chang, and Hsiang-Pang Li. Sparse reram engine: joint exploration of activation and weight sparsity in compressed neural networks. In *Proceedings of the 46th International Symposium on Computer Architecture*, pages 236–249. ACM, 2019.

[278] Zhewei Yao, Zhen Dong, Zhangcheng Zheng, Amir Gholami, Jiali Yu, Eric Tan, Leyuan Wang, Qijing Huang, Yida Wang, Michael Mahoney, and Kurt Keutzer. Hawq-v3: Dyadic neural network quantization. In *International Conference on Machine Learning*, pages 11875–11886. PMLR, 2021.

[279] Amir Yazdanbakhsh, Michael Brzozowski, Behnam Khaleghi, Soroush Ghodrati, Kambiz Samadi, Nam Sung Kim, and Hadi Esmaeilzadeh. Flexigan: An end-to-end solution for fpga acceleration of generative adversarial networks. In *FCCM*, 2018.

[280] Amir Yazdanbakhsh, Hajar Falahati, Philip J. Wolfe, Kambiz Samadi, Hadi Esmaeilzadeh, and Nam Sung Kim. GANAX: A Unified SIMD-MIMD Acceleration for Generative Adversarial Network. In *ISCA*, 2018.

[281] Amir Yazdanbakhsh, Ashkan Moradifirouzabadi, Zheng Li, and Mingu Kang. Sparse Attention Acceleration with Synergistic In-Memory Pruning and On-Chip Recomputation. In *MICRO*, 2022.

[282] Amir Yazdanbakhsh, Jongse Park, Hardik Sharma, Pejman Lotfi-Kamran, and Hadi Esmaeilzadeh. Neural Acceleration for GPU Throughput Processors. In *MICRO*, 2015.

[283] Amir Yazdanbakhsh, Choungki Song, Jacob Sacks, Pejman Lotfi-Kamran, Hadi Esmaeilzadeh, and Nam Sung Kim. In-DRAM Near-Data Approximate Acceleration for GPUs. In *PACT*, 2018.

[284] Deming Ye, Yankai Lin, Yufei Huang, and Maosong Sun. TR-BERT: Dynamic Token Reduction for Accelerating BERT Inference. 2021.

[285] Zihao Ye, Qipeng Guo, Quan Gan, Xipeng Qiu, and Zheng Zhang. Bp-transformer: Modelling long-range context via binary partitioning. *arXiv*, 2019.

[286] Geng Yuan, Payman Behnam, Zhengang Li, Ali Shafiee, Sheng Lin, Xiaolong Ma, Hang Liu, Xuehai Qian, Mahdi Nazm Bojnordi, Yanzhi Wang, and Caiwen Ding. Forms: fine-grained polarized reram-based in-situ computation for mixed-signal dnn accelerator. In *ISCA*, 2021.

[287] Ali Hadi Zadeh, Isak Edo, Omar Mohamed Awad, and Andreas Moshovos. Gobo: Quantizing attention-based nlp models for low latency and energy efficient inference. In *MICRO*, 2020.

[288] Sergey Zagoruyko and Nikos Komodakis. Wide Residual Networks. 2016.

[289] Manzil Zaheer, Guru Guruganesh, Kumar Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontanon, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, and Amr Ahmed. Big bird: Transformers for longer sequences. In *NeurIPS*, 2020.

[290] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *FPGA*, 2015.

[291] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks. In *FPGA*, 2015.

[292] Dongqing Zhang, Jiaolong Yang, Dongqiangzi Ye, and Gang Hua. Lq-nets: Learned quantization for highly accurate and compact deep neural networks. *arXiv preprint arXiv:1807.10029*, 2018.

[293] Jiaqi Zhang, Xiangru Chen, Mingcong Song, and Tao Li. Eager Pruning: Algorithm and Architecture Support for Fast Training of Deep Neural Networks. In *ISCA*, 2019.

[294] Jintao Zhang, Zhuo Wang, and Naveen Verma. 18.4 a matrix-multiplying adc implementing a machine-learning classifier directly with data conversion. In *ISSCC*, 2015.

[295] Shijin Zhang, Zidong Du, Lei Zhang, Huiying Lan, Shaoli Liu, Ling Li, Qi Guo, Tianshi Chen, and Yunji Chen. Cambricon-x: An accelerator for sparse neural networks. In *MICRO*, 2016.

[296] Yunqi Zhang, Michael A Laurenzano, Jason Mars, and Lingjia Tang. SMiTe: Precise QoS Prediction on Real-System SMT Processors to Improve Utilization in Warehouse Scale Computers. In *MICRO*, 2014.

[297] Guangxiang Zhao, Junyang Lin, Zhiyuan Zhang, Xuancheng Ren, Qi Su, and Xu Sun. Explicit sparse transformer: Concentrated attention through explicit selection. *arXiv*, 2019.

[298] Shuchang Zhou, Zekun Ni, Xinyu Zhou, He Wen, Yuxin Wu, and Yuheng Zou. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv*, 2016.

[299] Yanqi Zhou, Xuanyi Dong, Tianjian Meng, Mingxing Tan, Berkin Akin, Daiyi Peng, Amir Yazdanbakhsh, Da Huang, Ravi Narayanaswami, and James Laudon. Towards the Co-design of Neural Networks and Accelerators. In *MLSys*, 2022.

[300] Haishan Zhu and Mattan Erez. Dirigent: Enforcing qos for latency-critical tasks on shared multicore systems. In *ASPLOS*, 2016.

[301] Yuhao Zhu, Matthew Halpern, and Vijay Janapa Reddi. Event-based scheduling for energy-efficient qos (eqos) in mobile web applications. In *HPCA*, 2015.

[302] Yuhao Zhu, Daniel Richins, Matthew Halpern, and Vijay Janapa Reddi. Microarchitectural implications of event-driven server-side web applications. In *MICRO*, 2015.

[303] Neta Zmora, Guy Jacob, and Gal Novik. Neural network distiller, June 2018.

[304] Hui Zou and Trevor Hastie. Regularization and Variable Selection via the Elastic Net. *Journal of the royal statistical society: series B (statistical methodology)*, 2005.