UNIVERSITY OF CALIFORNIA

Los Angeles

Efficient Machine Learning Acceleration at the Edge

A dissertation submitted in partial satisfaction

of the requirements for the degree

Doctor of Philosophy in Electrical and Computer Engineering

by

Wojciech Piotr Romaszkan

2023

© Copyright by
Wojciech Piotr Romaszkan
2023

ABSTRACT OF THE DISSERTATION

Efficient Machine Learning
Acceleration at the Edge

by

Wojciech Piotr Romaszkan
Doctor of Philosophy in Electrical and Computer Engineering
University of California, Los Angeles, 2023
Professor Puneet Gupta, Chair

My thesis is a result of a confluence of several trends that have emerged in recent years. First, the rapid proliferation of deep learning across the application and hardware landscapes is creating an immense demand for computing power. Second, the waning of Moore's Law is paving the way for domain-specific acceleration as a means of delivering performance improvements. Third, deep learning's inherent error tolerance is reviving long-forgotten approximate computing paradigms. Fourth, latency, energy, and privacy considerations are increasingly pushing deep learning towards edge inference, with its stringent deployment constraints. All of the above have created a unique, once-in-a-generation opportunity for accelerated widespread adoption of new classes of hardware and algorithms, provided they can deliver fast, efficient, and accurate deep learning inference within a tight area and energy envelope.

One approach towards efficient machine learning acceleration that I have explored attempts to push a neural network model size to its absolute minimum. 3PXNet - Pruned, Permuted, Packed XNOR Networks combines two widely used model compression techniques:

binarization and sparsity to deliver usable models with a size down to single kilobytes. It uses an innovative combination of weight permutation and packing to create structured sparsity that can be implemented efficiently in both software and hardware. 3PXNet has been deployed as an open-source library targeting microcontroller-class devices with various software optimizations, further improving runtime and storage requirements.

The second line of work I have pursued is the application of stochastic computing (SC). It is an approximate, stream-based computing paradigm enabling extremely area-efficient implementations of basic arithmetic operations such as multiplication and addition. SC has been enjoying a renaissance over the past few years due to its unique synergy with deep learning. On the one hand, SC makes it possible to implement extremely dense multiply-accumulate (MAC) computational fabric well suited towards computing large linear algebra kernels, which are the bread-and-butter of deep neural networks. On the other hand, those neural networks exhibit immense approximation tolerance levels, making SC a viable implementation candidate.

However, several issues need to be solved to make the SC acceleration of neural networks feasible. The area efficiency comes at the cost of long stream processing latency. The conversion cost between fixed-point and stochastic representations can cancel out the gains from computation efficiency if not managed correctly. The above issues lead to a question on how to design an accelerator architecture that best takes advantage of SC's benefits and minimizes its shortcomings. To address this, I proposed the ACOUSTIC (Accelerating Convolutional Neural Networks through Or-Unipolar Skipped Stochastic Computing) architecture and its extension - GEO (Generation and Execution Optimized Stochastic Computing Accelerator for Neural Networks). ACOUSTIC is an architecture that tries to maximize SC's compute density to amortize conversion costs and memory accesses, delivering system-level reduction in inference energy and latency. It has taped out and demonstrated in silicon, using a 14nm fabrication process. GEO addresses some of the shortcomings of ACOUSTIC. Through the introduction of near-memory computation fabric, GEO enables a more flexible selection of

dataflows. Novel progressive buffering scheme unique to SC lowers the reliance on high memory bandwidth. Overall, my work tries to approach accelerator design from the systems perspective, making it stand apart from most recent SC publications targeting point improvements in the computation itself.

As an extension to the above line of work, I have explored the combination of SC and sparsity, to apply it to new classes of applications, and enable further benefits. I have proposed the first SC accelerator that supports weight sparsity - SASCHA (Sparsity-Aware Stochastic Computing Hardware Architecture for Neural Network Acceleration), which can improve performance on pruned neural networks, while maintaining the throughput when processing dense ones. SASCHA solves a series of unique, non-trivial challenges of combining SC with sparsity. On the other hand, I have also designed an architecture for accelerating event-based camera object tracking - SCIMITAR. Event-based cameras are relatively new imaging devices which only transmit information about pixels that have changed in brightness, resulting in very high input sparsity. SCIMITAR combines SC with computing-in-memory (CIM), and, through a series of architectural optimizations, is able to take advantage of this new data format to deliver low-latency object detection for tracking applications.

The dissertation of Wojciech Piotr Romaszkan is approved.

Mani B. Srivastava

Sudhakar Pamarti

Anthony J. Nowatzki

Puneet Gupta, Committee Chair

University of California, Los Angeles

2023

To Camila.

TABLE OF CONTENTS

1 Introduction 1

1.1 Model Compression: Better, Faster, Smaller 2

1.1.1 Fewer Bits - Quantization & Binarization 2

1.1.2 Alternate Number Representations 3

1.1.3 Fewer Values - Sparsity & Pruning 4

1.1.4 Edge Models & Libraries 6

1.2 Stochastic Computing - Processing with Random Streams 6

1.2.1 Number Representation 7

1.2.2 Multiplication and Accumulation 8

1.2.3 Stochastic Neural Network Functions 10

1.3 Domain-Specific Acceleration - Breaking the Shackles of Generality 11

1.4 Dissertation Outline 12

2 3PXNet - Fewer Bits, More Zeros **14**

2.1 A Case for Sparse XNOR Networks 15

2.2 The 3PXNet Approach 17

2.2.1 Challenges in pruning XNOR networks 17

2.2.2 Pruning a packed XNOR network 19

2.2.3 Training 3PXNets 21

2.3 Implementing 3PXNet 24

2.3.1 Fully-Connected Layers 25

2.3.2 Convolutional Layers 28

2.3.3 Fused kernels 32

2.3.4 ARM NEON Support 33

2.3.5 Binarization of the First Layer 34

2.4 Experimental Setup 34

2.4.1 Platforms 35

2.4.2 Benchmarks 35

2.4.3 Baseline 36

2.5 Results and Discussion 36

2.5.1 Accuracy & Model Size 36

2.5.2 Performance & Energy 39

2.6 Distinction between 3PXNet and Ternary Networks 42

2.7 Conclusion 42

3 ACOUSTIC - Accelerator Built on Randomness 44

3.1 Introduction 45

3.2 ACOUSTIC Optimizations for DNNs 48

3.2.1 Split-Unipolar Representation 48

3.2.2 OR-based Scaling-Free Accumulation 49

3.2.3 Computation Skipping using Stochastic Average Pooling 51

3.3 ACOUSTIC Architecture 51

3.3.1 Understanding SC Benefits 52

3.3.2 Accelerator Architecture 55

3.3.3 Control 65

3.3.4 Evaluated ACOUSTIC Architectures 67

3.4 Evaluation & Results . . . 69
3.4.1 Evaluation Methodology . . . 69
3.4.2 ACOUSTIC Accuracy . . . 70
3.4.3 Runtime Configurable Precision . . . 71
3.4.4 Area & Power Breakdown . . . 71
3.4.5 Performance Comparisons . . . 73
3.5 FPGA Evaluation . . . 75
3.6 Demonstration Chip . . . 76
3.6.1 Architecture . . . 77
3.6.2 SC Computation . . . 80
3.6.3 Computation Mapping . . . 80
3.6.4 Evaluation & Results . . . 83
3.7 Related Work . . . 86
3.7.1 Deep Learning using Stochastic Computing . . . 86
3.7.2 Approximate and Programmable Precision Accelerators . . . 88
3.8 Conclusion . . . 89
4 GEO - Pushing Stochastic Computing Further . . . 91
4.1 Introduction . . . 92
4.2 Stochastic Stream Generation Optimizations . . . 94
4.2.1 Co-optimized Shared Generation and Training . . . 94
4.2.2 Progressive Stochastic Stream Generation . . . 95
4.3 Stochastic Computing Execution Optimizations . . . 97
4.3.1 GEO Architecture . . . 97

4.3.2 Partial Binary Accumulation . . . 99
4.3.3 Near-Memory Computation . . . 101
4.3.4 Pipeline Optimizations . . . 102
4.4 Evaluation & Results . . . 103
4.4.1 Evaluation Methodology . . . 103
4.4.2 GEO Accuracy Comparisons . . . 105
4.4.3 Performance Impact of GEO Enhancements . . . 105
4.4.4 GEO Performance Compared . . . 107
4.5 Conclusion . . . 108

5 SASCHA - Combining Randomness with Sparsity . . . 110
5.1 Introduction . . . 111
5.2 Motivation . . . 112
5.3 SASCHA Sparse SC PE . . . 114
5.3.1 Sparse PE Design Objectives . . . 114
5.3.2 G:C Sparse PE . . . 115
5.3.3 Multi-Group Sparse SC PE . . . 119
5.3.4 SASCHA PE Analytical Model . . . 122
5.3.5 Parallel Stream Processing . . . 124
5.4 SASCHA Architecture . . . 127
5.4.1 SASCHA Accelerator . . . 127
5.4.2 SASCHA Asynchronous Scheduler . . . 130
5.4.3 Memory Organization . . . 133
5.5 Bit-Slicing Weights . . . 134

5.6 Evaluation & Results 139
5.6.1 SASCHA Accuracy 139
5.6.2 Performance Results 141
5.7 Related Work 146
5.7.1 Sparse Accelerators. 146
5.7.2 Stochastic Computing Accelerators. 147
5.8 Conclusion 147

6 SCIMITAR: Event-Based Tracking with Stochastic Compute-In-Memory 149
6.1 Introduction 150
6.2 Motivation 152
6.2.1 Event-Based Cameras 152
6.2.2 Event-Based Data Processing 154
6.2.3 Stochastic Computing In-Memory 159
6.3 SCIMITAR Implementation 161
6.3.1 Stochastic Compute-in-Memory Macro with In-Situ SNG 161
6.3.2 In-Situ Stochastic Number Generator 162
6.3.3 In-Memory Stochastic MAC Unit 164
6.3.4 Event-Based SCIM Accelerator Architecture 166
6.3.5 Multi-Level Early Termination 175
6.4 Evaluation 178
6.4.1 Accuracy 178
6.4.2 Hardware Evaluation 179
6.5 Conclusion 182

7 Conclusion 183
7.1 Overview of Contributions 183
7.1.1 3PXNet 183
7.1.2 ACOUSTIC & GEO 184
7.1.3 SASCHA 185
7.1.4 SCIMITAR 186
7.2 Directions for Future Work 186
7.2.1 Exploration of Stochastic Computing Accelerators 186
7.2.2 Analog Stochastic Computing 187
7.2.3 Extending 3PXNet 188
References 189

LIST OF FIGURES

1.1 Examples of unipolar (a) and bipolar (b) SC representation, and a unipolar Stochastic Number Generator Circuit (SNG) with an AND gate SC multiplication (c). 7

1.2 RMS error of the bit stream representation for a given value a), and area [μm^2] comparison between different SC MACs, depending on the number of inputs b). 9

2.1 Storage requirements for Dense, NP and NPP XNOR "small" MNIST MLP for varying levels of sparsity. 19

2.2 Pruning with packing constraint of 4 bits a) without permutation, b) with permutation 21

2.3 Comparison of training results with and without permutation for different sparsities. 22

2.4 A schematic view of a fully-connected layer with NI inputs and NO kernels. . . 24

2.5 A schematic view of a padded convolutional layer followed by pooling operation. 25

2.6 256x4 fully-connected layer weight and index storage with 75% sparsity (NP=2x 32-bit packs per output). 28

2.7 3x3x32 kernel packed into depth-first binarized vectors. 29

2.8 Dense and 3PXNet (93.75% sparsity) speedups for kernel as an outer (K-YX) and inner (YX-K) loop for different VGG-16D [1] layers, normalized to dense K-YX. 30

2.9 3x3x32 Convolution kernel weight packs and indices with KL=3 active packs. . 31

2.10 Convolution splitting into padded and non-padded regions for efficient computation. 33

2.11 A 2x2 fused Max Pooling followed by threshold Batch Normalization. 34

2.12 Accuracy vs. Memory tradeoff compared to eBNN and dense XNOR. 37

2.13 Accuracy comparison between sparse 8-bit network and 3PXNet, for MNIST (a) and CIFAR-10 (b). 40

3.1 Circuit level support for unipolar, temporal split-unipolar, and spatial split-unipolar representations (a) and an example of 2-wide split-unipolar MAC temporarily-(b) and spatially-unrolled c). 49

3.2 Accuracy comparison between MUX and OR a) and comparison of approximation methods for OR accumulation b). 50

3.3 Normalized MAC energy (a) and area (b) for 8-bit fixed-point and 256-long, unipolar SC implementations in TSMC 28nm node with 200MHz clock, with different data reuse patterns. Normalized MAC energy for 256-wide MAC when intermediate results are converted to binary (c). 53

3.4 Block diagram of the proposed ACOUSTIC accelerator (a), and the hierarchical organization of the compute engine with parts of the kernel and activation tensors covered by each level of hierarchy (b). 56

3.5 Convolving a 1x128x32 input slice with a 1x3x32 kernel to compute a 1x16x1 partial output slice a). Extension across multiple arrays to compute a 1x128x1 output slice b). Configuration for a 1x64x64 input tensor. 59

3.6 Processing two successive output rows sequentially (a), processing multiple kernels at the same time, with output transposition (b). 61

3.7 Extending kernel size up to 6x6 by coupling adjacent rows (a), enabling padding through row scheduling for height (b) and configurable shifting fabric before array inputs for width (c). 62

3.8 Latency of processing a convolutional layer with 16x16x512 inputs and 512 3x3x512 kernels and pre-loading 512 3x3x512 kernels for the subsequent layers using different clock frequency and external memory interfaces, using temporarily-unrolled 256-long split-unipolar streams. 68

3.9 Accuracy and performance at different stream lengths for the CIFAR-10 CNN on ACOUSTIC ULP. Labeled points are pareto points with the numbers representing stream lengths in 1st layer, 2nd & 3rd layer, and 4th layer. 72

3.10 Area breakdown for ACOUSTIC LP (a) and ULP (b) and power breakdown for ACOUSTIC LP (c) and ULP (d). 72

3.11 Overall accelerator architecture. 78

3.12 SC split-unipolar MAC and stochastic average pooling (top). Precision-latency trade-off using different stream lengths (bottom). 79

3.13 Mapping of convolutional layers in MAC rows, and memory/stream generation amortization through data reuse. Shift-register organization emulating sliding window (left), MAC block with input/weight reuse and padding support (center), block organization implementing different levels of reduction (right). 81

3.14 Normalized ratio of MAC to memory accesses and stream generations compared to fixed-point designs. Accelerator area scaled to 14nm is included. 81

3.15 Model deployment pipeline. 83

3.16 Die shot and specifications. 84

3.17 Accuracy, latency (left), and energy (right) on the MNIST (top), SVHN and CIFAR-10 (bottom) datasets. 85

3.18 Area (left) and power (right) breakdown, compared to [2, 3, 4]. 86

3.19 Peak energy efficiency at different stream lengths. 86

4.1 Accuracy vs. sharing for TRNG and LFSR-based random number generation. . 95

4.2 Accuracy comparison between normal generation and progressive generation performing a multiplication of two uniformly sampled inputs. RMS Error is multiplication error compared to an 8-bit integer. 96
4.3 Normal SNG (a) and progressive stream generation (b). 97
4.4 Overall SC accelerator architecture block diagram. with breakdowns of the MAC row (left) and output converter (right) modules (a). Fixed 8-bit maximum length LFSR (b), and configurable 8- or 7-bit maximum length LFSR (c). 98
4.5 Area comparison for different hardware implementations of SC-based MAC units for different kernel sizes and different levels of partial binary accumulation. . . . 100
4.6 Area, energy and latency for different GEO configurations (normalized to Base-128,128). 107
5.1 Sparse PE with group size G and capacity C (a). Decomposing 3 arbitrary parameter groups of size $G = 4$, into groups satisfying the capacity requirement of $C \leq 2$ (b). 116
5.2 Sparse GEO-style SC PE with group size G and capacity C (a). Split-Unipolar [5] logic is omitted case for readability. Area breakdown of fixed-point (left) and GEO SC (right) sparse PEs with G=4, and C=2 (b). 118
5.3 Area of sparse and dense SC PEs, given different group sizes and capacities, for GEO (a), GEO with full binary accumulation (b), and uGEMM (c) style SC. . . 120
5.4 Ideal ratio of dense to sparse storage cost for different PE group sizes, and sparsity levels. Gray line shows the break-even point between sparse and dense storage. . 121
5.5 Single-group sparse PE with a group size G, capacity C and dot product width K (a), and a throughput-equivalent multi-group sparse PE with L groups. . . . 122

5.6 Multi-group $K = 16$ sparse SC PE iteration overheads normalized to dense PE area (GEO-style), for different group size G and capacity C, at different sparsity levels, estimated using the analytical model (a). Gray line shows the latency break-even point with a dense PE. Iteration overhead difference between the model and an ideal scheduler described in Section 5.4 on the CIFAR-10 TinyConv network, for a PE with $G = 4$ (b). 125

5.7 Sparse SC PE with group size G, capacity $C = 1$, and $P = 2$ parallel streams. Split-unipolar accumulation fabric is omitted for readability. 126

5.8 Total area of a 32x32 array of $K = 32$ GEO (a), GEO+ (b) and uGEMM (c) PEs, dense and sparse, with different stream parallelism factors. $C = 1$ for all sparse configurations. 127

5.9 SASCHA accelerator architecture block diagram. Partial sum output connections were omitted for readability. 129

5.10 Three schedules of 5 partial filters, with $K = G = 4$ and $C = 1$, on an architecture with $M = 4$ rows: dense synchronous (a), sparse synchronous (b) and sparse asynchronous (c). Crossed out boxed indicate compute underutilization. 131

5.11 Iteration overhead using different sparse scheduling methods (a) and different group sizes and capacity using the sparse asynchronous scheduler (b). 133

5.12 Number of memory accesses in bytes for the convolutional layers of CIFAR-10 TinyConv network at 90% sparsity, depending on the choice of scheduling and dataflow. All results for $M = 32, N = 32$, and $K = 32$. Sparse results for $G = 8, C = 1$, and $P = 8$. 135

5.13 Overall, MSB and LSB sparsity for (a) and reduction in sliced multiplication area x delay cost relative to non-sliced cost for SC and fixed-point (b), at different pruning levels for CIFAR-10 TinyConv. 136

5.14 SC unipolar multiplication a), and sliced multiplication b). 137

5.15 Accuracy of CIFAR-10 VGG-11 with different sparsity levels. 0% sparsity means no sparsity constraint. 139

5.16 SASCHA CIFAR-10 Top-1 accuracy with dense and sparse networks. 140

5.17 SASCHA ImageNet Top-5 accuracy with dense and sparse networks. 141

5.18 Area (a) and power (b) breakdown of SASCHA GEO 8/1/8. 143

6.1 Spinning Marker. This image shows event data generated from tossing a spinning whiteboard marker into the air with a cluttered background. The stationary background has disappeared so it is easy to see the moving objects. The white events are positive events indicating an increase in brightness and the blue events signal a decrease in brightness. 153

6.2 Analytical model parameters, for frame- (top), ROI- (middle), and event-based (bottom) processing. 156

6.3 Input (top) and output (middle) memory accesses, and MAC count (bottom), with varying event count. 157

6.4 Input memory accesses (top), and MAC count (bottom), with varying ROI count. 158

6.5 256-wide SCIM MAC structure. 160

6.6 SCIM Macro architecture. 161

6.7 Split-unipolar stochastic representation (top) and in-situ stochastic number generator (SNG) circuit (bottom). 163

6.8 Interleaved Signed SC MAC unit. 165

6.9 SCIM Unit with 32 MAC reuse and SCIM slice 166

6.10 SCIMITAR architecture block diagram. 167

6.11 Transposed memory layout for 6-bit (top) and 1-bit (bottom) input data. 169

6.12 ROI memory requirements for different compression schemes. 170

6.13 Channel load skipping and half-row multiplexing using partitioned input SRAM. 171

6.14 Deserializing staging buffers with zero bit indicator. 172

6.15 Time channel overlap support using circular staging buffers. 174

6.16 Impact of proposed optimizations on computational energy efficiency of the SCIMITAR architecture. Efficiency calculated on 99% sparse input data assuming 64-bit long SC streams. 175

6.17 Schematic implementation of 3-level early termination. 177

6.18 SCIM Macro timing (top), and energy breakdown (bottom). 180

6.19 Energy breakdown of the SCIMITAR components for dense (outer circle), 90% sparse (middle circle), and 99% sparse (inner circle) workloads. Energy is calculated using 64-long stochastic streams. 181

LIST OF TABLES

2.1 Hardware platforms used for the runtime experiments. 15

2.2 Weight storage requirements of different networks depending on precision. . . . 16

2.3 Benchmark models and datasets 20

2.4 Accuracy and network size (KB, in brackets) comparison. 38

2.5 Runtime (ms) and energy (mJ, in brackets) for MNIST networks. A dash indicates a given model could not fit in memory. 41

2.6 Runtime (ms) and energy (mJ, in brackets) for CIFAR-10/SVHN/Speech networks. A dash indicates a given model could not fit in memory. 42

3.1 Accuracy comparison between different pooling methods. 52

3.2 ACOUSTIC control modules and their respective instructions. 66

3.3 Accuracy comparisons. 70

3.4 Performance comparison between ACOUSTIC LP and other fixed-point and stochastic accelerators. 74

3.5 Performance comparison between ACOUSTIC ULP, MDL CNN [6] and Conv-RAM [7] on convolutional layers of LeNet-5 and CIFAR-10 CNN. 75

3.6 FPGA utilization and performance comparison between ACOUSTIC ULP and other convolutional neural network accelerators. ACOUSTIC performance is for stream lengths in range of 32 to 256-bits. 77

3.7 Datasets and models used in evaluation. Model sizes are limited by available on-chip memory. 84

3.8 Comparison table. 87

4.1 Accuracy comparison with fixed-point, other SC implementations and so on. . . 106

4.2 Comparison between GEO ULP and fixed-point and neuromorphic implementations. Numbers are scaled to 28nm. 108

4.3 Comparison between GEO LP and fixed-point and SC implementations. Numbers are scaled to 28nm. 109

5.1 RMSE of unipolar multiplication with and without bit-slicing, w.r.t. floating-point precision, for different stream lengths (1000 trials). LSB stream length is 8. 138

5.2 Area [mm^2], power [mW], throughput [Fr/s] and energy-efficiency [Fr/J] for different accelerators, models and datasets, and sparsity. 144

5.3 Weight compression ratio for different SASCHA configurations, networks, and sparsity levels. 146

6.1 Analytical event-based tracking performance model. *Event*, *frame*, and *ROI* refer to event-, frame-, and ROI-based processing, respectively. 156

6.2 Average ROI processing latency in cycles, with and without early termination for different stream leangths. 178

6.3 Accuracy metrics of approximate computation in tracking applications using Gabor filters. 179

ACKNOWLEDGMENTS

As with anything worth doing in life, this dissertation is a result of years of hard work involving successes and moments of profound satisfaction but also struggles and disappointments. Most importantly, it is not a result of an individual toiling in isolation - I owe so much to so many people, it would be impossible to list them all here. Nevertheless, I will try my best, and I apologize to anyone who was omitted.

First and foremost, I want to thank my advisor, Professor Puneet Gupta. He showed faith in me by accepting me into his group, gave me all the support and resources I needed, and pushed me to accomplish my best. The volume and quality of the work presented here would not have been possible without his drive for excellence. Professor Gupta consistently shows the highest level of curiosity and tenacity when approaching any new topic or problem, something I aspire to emulate. Thanks to him, I have been exposed to a vast breadth of subjects that has expanded my horizons as a researcher, engineer, and technology enthusiast. I am truly grateful for his guidance and the opportunity to work with him.

I would also like to thank the other members of my doctoral committee: Professor Sudhakar Pamarti, Professor Mani Srivastava, and Professor Tony Nowatzki, for both teaching and research guidance. In particular, Professor Pamarti has been involved in the vast majority of the work presented here. He provided invaluable guidance on topics such as analog design, signal processing, and probability, without which this dissertation would hot have been possible. Professor Srivastava facilitated many productive discussions with the members of his group, and Professor Nowatzki introduced me to many new topics in computer architecture.

I would like to thank my internship managers, mentors, and coworkers. At Amazon's Annapurna Labs, I worked under the guidance of Patricio Kaplan and Paul Meyer, who introduced me to the world of cloud machine learning acceleration. At Facebook (now Meta), I had the pleasure of working with Rudy Tan and Piyush Agarwal, who exposed

me to the intricacies and challenges of designing AR/VR devices. Both internships were extremely rewarding and expanded my professional horizons. I would also like to thank my undergraduate and master's theses advisors: the late Dr. Jerzy Kasperek, and Dr. Paweł Rajda at the AGH University of Science and Technology. They introduced me to the field of research and gave me a chance to work on fascinating projects. I also want to thank my former mentors and coworkers, in particular Dilip Bansal (now with Intel) and Ali Rabbani (now with Apple) at Imagination Technologies, Dr. Toshiyuki Ikeda at NEC Corporation, and Dr. Marcin Szczurkowski at Elsta Elektronika. Learning from them has laid the foundations that made me ready to undertake doctoral studies.

Next, I would like to thank my collaborators, NanoCAD labmates, and fellow students: Tianmu Li, Dr. Saptadeep Pal, Dr. Irina Alam, Dr. Mark Gottscho, Dr. Wei-Che Wang, Shurui Li, Alexander Graening, Rhesa M. Ramadhan, Jiyue Yang, Vinod Kurian Jacob, Albert Lee, Yoo-Jin Chae, Dr. Steven Moran, Dr. Sumeet Singh, Professor Ankur Mehta, Swapnil Saha, Dr. Sandeep Singh, Rahul Garg, Trevor Black, Erin Askounis, Brian Zutter, and many more. They have all made my time at UCLA much more fulfilling and enjoyable. I also want to acknowledge my mentees: Tristan Melton, Feiqian Zhu, Siddharth Nandy, and Ravit Sharma. I might not have been the best mentor, but you all taught me a lot, and I hope I made a positive contribution to your career trajectories. My enormous gratitude goes out to UCLA Electrical and Computer Engineering staff, especially the irreplaceable Deeona Columbia, but also Jose Cano, Tricia Senate, and many others. Their efforts and dedication have been invaluable to this department, and me personally.

I could not have gotten here without the love, support, and encouragement of my parents, sister, and my maternal grandparents. Through good or bad, you made sure I was able to pursue my interests and made me the person I am today. My thanks also go to my friends, especially the ones I sometimes do not see for years, and then pick up exactly where we left off: Ewa Gadocha-Cios, Mateusz Tarnawa, and George Hawkins.

Finally, and most importantly, I would like to thank Dr. Camila Cendra, my wonderful

partner, who has been through it all with me. From being strangers in a strange land, through the highs and lows, the long distance, the longer distance, the pandemic, the two PhDs, across three continents, for over nine years, you have been my rock, my best friend, my soulmate.

Copyrights and Re-use of Published Material

This dissertation contains significant material that has been previously published or is intended to be published in the future. Chapter 2 (3PXNet) contains material that was published in [8]. Chapter 3 (ACOUSTIC) includes material published in [5, 9]. Chapter 4 (GEO) appeared in [10]. Chapter 5 (SASCHA) appeared in [11]. Chapter 6 (SCIMITAR) is part of a paper which is under submission.

The copyrights on published research that re-appears in this dissertation is with either the IEEE and/or the ACM where appropriate. The respective copyright agreements allow for derivative works by the author with attribution, so no explicit permission was required for inclusion of material in this dissertation. The titles of each chapter have been changed somewhat to differentiate them from the published versions of the respective manuscripts where applicable.

Work explored in Chapters 2, 3, 4, and 5 has been done jointly with my labmate Tianmu Li, who has focused on the accuracy and training side, while I worked on architecture and implementation. Rahul Garg, Tristan Melton, and Jiyue Yang have contributed to the chip tapeout described in Chapter 3. Work described in Chapter 6 has been co-authored with Jiyue Yang, Vinod Kurian Jacob, and Alexander Graening. The former two focused on the analog circuit implementation, while the latter on the application side of tracking. Professor Puneet Gupta has contributed as a principal investigator (PI) to all of the work in this thesis, while Professor Sudhakar Pamarti served as a co-PI for the work described in Chapters 3, 4, 6.

Some of the work in my PhD that was conducted in collaboration with other individuals (where I contributed, but did not lead) are not included in the body of this dissertation.

Work presented in chapters 3, 4, 5, and 6 has been sponsored by the Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) under agreement number FA8650-18-27867. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Air Force Research Laboratory (AFRL) and Defense Advance Research Projects Agency (DARPA) or the U.S. Government. Part of my PhD has been sponsored by the Fulbright Program under a Graduate Student Award, as well as through fellowships granted by the Electrical and Computer Engineering Department at UCLA.

VITA

2012	B.Eng. (Electronics and Telecommunication), AGH University of Science and Technology, Kraków, Poland.
2013	M.Eng., Honors (Electronics and Telecommunication), AGH University of Science and Technology, Kraków, Poland.
2013-2014	Vulcanus in Japan Fellowship
2014-2017	Hardware Engineer, Imagination Technologies Ltd.
2017	Fulbright Graduate Student Award
2017	Ph.D. Departmental Fellowship, ECE Department, UCLA
2018	Ph.D. Preliminary Examination Fellowship, ECE Department, UCLA
2020	Digital Design Intern, Facebook Inc. (now Meta Platforms Inc.)
2022	ASIC Design Intern, Amazon Web Services Inc.

PUBLICATIONS

Wojciech Romaszkan, Tianmu Li, and Puneet Gupta, "3PXNet: Pruned-Permuted-Packed XNOR Networks for Edge Machine Learning," in *ACM Transactions on Embedded Computing Systems (TECS)* 19, no. 1, pp. 1-23, 2020.

Wojciech Romaszkan, Tianmu Li, Tristan Melton, Sudhakar Pamarti, and Puneet Gupta, "ACOUSTIC: accelerating convolutional neural networks through or-unipolar skipped stochastic computing," in *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, Lausanne, Switzerland, pp. 768-773, 2020.

Tianmu Li, **Wojciech Romaszkan**, Sudhakar Pamarti, and Puneet Gupta, "GEO: Generation and Execution Optimized Stochastic Computing Accelerator for Neural Networks," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, Lausanne, Switzerland, pp. 1-6, 2021.

Shurui Li, **Wojciech Romaszkan**, Alexander Graening, and Puneet Gupta, "SWIS - Shared Weight bIt Sparsity for Efficient Neural Network Acceleration," in *First International Research Symposium on Tiny Machine Learning (tinyML)*, Burlingame, USA, pp. 1-8, 2021.

Wojciech Romaszkan, Tianmu Li, Rahul Garg, Jiyue Yang, Sudhakar Pamarti, and Puneet Gupta, "A 4.4-75-TOPS/W 14-nm Programmable, Performance-and Precision-Tunable All-Digital Stochastic Computing Neural Network Inference Accelerator," in *IEEE Solid-State Circuits Letters 5*, pp. 206-209, 2022

Wojciech Romaszkan, Tianmu Li, and Puneet Gupta, "SASCHA—Sparsity-Aware Stochastic Computing Hardware Architecture for Neural Network Acceleration," in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 41*, pp. 4169-4180, no. 11, 2022

Jiyue Yang, Tianmu Li, **Wojciech Romaszkan**, Puneet Gupta, and Sudhakar Pamarti, "A 65nm 8-bit All-Digital Stochastic-Compute-In-Memory Deep Learning Processor," in *2022 IEEE Asian Solid-State Circuits Conference (A-SSCC)*, pp. 10-11, 2022

CHAPTER 1

Introduction

In the last decade, the computing landscape has been engulfed by chaos. The looming deprecation of Moore's Law has created a demand for new devices, architectures, and algorithms to deliver the steady stream of performance improvements on which the markets have relied for decades [12]. At the same time, the rapid emergence of deep learning has created a massive demand for domain-specific computational power across both cloud and edge computing [13, 14, 15]. The existing infrastructure struggles to satisfy this demand, particularly at the edge, where deep learning's hunger for FLOPs and MBs goes against tight latency, energy, and privacy constraints [16]. Those conditions have created a perfect storm of opportunity for introducing new hardware architectures, especially at the edge, where silicon real estate is traditionally very scarce. Indeed, smartphone SoC with various "neural engines" are proliferating the markets as we speak [17].

Coupled with the above circumstances is the fact that deep learning algorithms have been shown to be incredibly approximation tolerant [18, 19, 20]. This tolerance has opened a pandora's box of new and old techniques, such as pruning or quantization, and hardware approaches, including spiking, neuromorphic, in-memory, and stochastic computing, among others [21, 22, 23, 24, 2, 25, 26]. What they all have in common is a promise of driving down deep learning model sizes and computational costs, enabling their widespread deployment in edge devices. The work in this dissertation explores, combines, and evolves those techniques to enable the development of future highly-compact and highly-efficient architectures and algorithms for deep learning.

The remainder of this Section provides a brief overview of the topics that form the foundation of this dissertation. Section 1.1 provides an introduction to neural network model compression, including quantization and pruning, as well as existing methods of model deployment on edge devices. Section 1.2 introduces stochastic computing and its intricacies. Section 1.3 discusses the rapidly growing field of domain-specific accelerators, particularly those targeting neural networks. Finally, Section 1.4 summarizes the work presented in this dissertation.

1.1 Model Compression: Better, Faster, Smaller

Ever since the staggering success of AlexNet in the ImageNet visual recognition challenge, deep learning models have been evolving at a rapid rate [27, 28]. Vision models based on convolutional layers quickly reached tens of giga-operations (GOPS) and tens if not hundreds of megabytes (MBs), in size [29, 30, 1, 31]. With complexity, their capabilities grew, becoming a force multiplier for edge devices, like smartphones or sensors [32, 33, 34]. However, this growing demand for resources made deploying those models restrictive, if not outright prohibitive, on those same devices [16]. And thus, a simple question - "how do we make the models smaller?" has preoccupied countless researchers for the better part of the last decade, yielding many ingenious solutions [35]. This Section aims to present a high level summary of those. A general introduction to deep neural networks is omitted for brevity, but readers not familiar with the topic can refer to [29, 36].

1.1.1 Fewer Bits - Quantization & Binarization

One of the first crucial insights into model compression was the realization that 32-bit floating-point precision, ubiquitous in scientific computing, was not necessary for highly redundant and error resilient deep learning models [14, 2]. Multiple works have shown that decreasing precision of underlying computation through quantization does not affect accu-

racy while significantly improving storage and runtime [37, 38, 39, 40, 19, 14]. A quick progression of lower precision models and libraries followed - 16-bit floating point [41], 16-bit fixed-point [2], 8-bit fixed-point [42], and below [19, 42, 24]. To reduce computational complexity to its absolute limit, researchers have proposed binarization, which restricted all values to -1 and +1 [43, 44, 45, 18, 46]. Those Networks are commonly referred to as XNOR-Nets, because multiplication between binarized values can be implemented using a bitwise XNOR operation.

The promise of significant performance and storage improvements given by XNOR-Nets has resulted in multiple software and hardware implementations. Umuroglu et al. have created FINN [47], a framework for binarized FPGA accelerators, further expanded to support larger models by Fraser et al. [48]. Other binarized accelerators have been proposed, both targeting FPGAs [49, 50, 51, 52], ASIC [53, 54, 55, 56], and in-memory compute [57, 58]. Yang et al. [59] have developed BMXNet, an extension of MXNet [60] based on binarized GEMM kernel. Depth-first binarized convolution implementations have also been shown for both CPUs and GPUs [61, 62, 63].

1.1.2 Alternate Number Representations

Besides quantization, researchers have explored alternative ways of representing numbers. One of them is the so-called *unary* computing. Compared to *binary* numbers, like fixed- or floating-point, where each bit has a positional significance, unary numbers are streams of bits with identical significance [64]. There are two types of unary numbers: rate- and temporal-coded. In the former, the value is encoded in a frequency of an event, e.g., the number of 1s in a stream. Rate-coded unary computation is commonly known as stochastic computing (SC). Since SC is a major focus of this dissertation, a more comprehensive introduction is provided in Section 1.2. Temporal coding encodes the information into the timing of a signal transition, with a stream consisting of a series of 1s followed by a series of 0s. Temporal coding can be performed within the stream generator, using the so-called thermometer coding

[65], or by designing paths with different delays, referred to as race-logic [66]. Compared to rate coding, temporally-coded numbers either do not have essential functional units or those units are highly inefficient. Further, temporal coding lacks many opportunities for improving accuracy possible in SC [67].

Other number representation methods are tied to the hardware implementation. A large body of work tries to imitate the behavior of the human brain through spiking neuron connectivity [26, 68, 69, 70]. Spiking hardware, commonly referred to as neuromorphic, represents data using the timing or frequency of the pulses, instead of their amplitude. This can be implemented in the digital or analog domain. Examples of the former include IBM TrueNorth [26], the SpiNNaker Project [68], and Bluehive [71]. Analog spiking hardware examples are: Neurogrid [69], HICANN [70], and CAVIAR [72]. While neuromorphic hardware has potential advantages, like energy efficiency, and on-device training using synaptic plasticity, they are also notoriously hard to train and scale [73]. Pulse width modulation (PWM) has also been used as a way of representing values for more efficient analog hardware implementations [6].

There are also purely digital alternative representations. Posit format has been proposed as an substitute for floating-point, offering variable-width exponent and fraction fields, higher dynamic range, and more efficient hardware implementations [74]. While some devices offer posit support, it has not yet been broadly adopted [75, 76]. Apart from that, various form of non-linear quantization have been proposed beyond fixed-point, often requiring custom software or hardware support [77, 24, 78].

1.1.3 Fewer Values - Sparsity & Pruning

The other realization foundational to model compression was that in computation based on linear algebra, multiplications involving a zero will not contribute to the final result, hence can be skipped [79]. The underlying computation then becomes *sparse*. While it applies to both weights and activations, the opportunities and mechanisms involved are

different. Zero-valued weights are enabled through the process of *pruning* the model, which involves removing connections in the model that are deemed inconsequential. It was first proposed over 20 years ago as a way of improving generalization and reducing computational complexity for both training and inference [22, 80]. Since activations are dynamic, they cannot be pruned statically in the same manner. However, activations can be effectively sparsified through the use of the popular rectified linear unit (ReLU) activation function [79, 81]. Once weights, activations, or both, are made sparse, in theory, it is possible to skip both storage and computation involving zero values. Recently, Han et. al. [82] have shown over 10x compression on popular network models with no increase in error rates. By further coupling pruning with quantization and efficient coding in a scheme called Deep Compression, they achieved up to 49x size reduction [83]. However, deploying pruned models on highly-parallel architectures has proven problematic due to the storage overhead and irregular memory access patterns of sparse matrix multiplication [21, 84].

To make pruning more regular, multiple forms of "structured" pruning have been proposed. Lebedev and Lempitsky [85] proposed group-wise sparsification. Foroosh et. al. [86] hard-coded the sparsity patterns into the source code, achieving up to 6.88x speedup on CPUs. Anwar et. al. [87, 88] explored different granularities of pruning: feature map, kernel, and intra-kernel and introduce kernel strided sparsity. Sredojevic et. al. [84] have proposed an algorithmic way of inducing regularity in sparse networks. Yu et. al. [21] have developed a hardware-aware pruning method called Scalpel, which matches the coarseness of pruning to the parallelism of the underlying hardware. Wang et. al. [89] have used structured sparsity in unrolled kernels after im2col conversion. Pruning has been successfully exploited in custom accelerators by using compressed storage, skipping memory accesses, gating computation, and exploiting novel dataflows [13, 90, 79, 19]. Crossbar-aware pruning has also been proposed given the recent emergence of analog crossbar-based accelerators [91].

1.1.4 Edge Models & Libraries

Edge Machine Learning inference on embedded platforms has been explored in recent years as a way to remove the communication energy and latency involved in offloading it to the servers. Due to severe memory and energy constraints of such devices, various model compression techniques have been used to make such applications feasible. Compressed Neural Network models like SqueezeNet [92], ShuffleNet [93], and MobileNet [94] have been developed, specifically targeting low memory footprints. Lai et. al. [95], have developed CMSIS-NN, a software library for ARM Cortex-M microcontrollers with 8- and 16-bit fixed-point support. Microsoft is developing EdgeML, a Machine Learning library containing algorithms optimized for low storage, energy and latency [96, 97]. Google has create TensorFlow Lite, and extension of the TensorFlow framework targetting microcontroller class devices [98]. Other examples include Apache microTVM, STM Cube.AI, Synopsys embARC, and MIT MCUNet [16, 99].

1.2 Stochastic Computing - Processing with Random Streams

Stochastic Computing (SC), introduced by [100] in the 1960s, is a number representation system. It uses a proportion of 1's in a binary stream to represent fractional numbers. Compared to conventional fixed- or floating-point formats, SC makes it possible to implement certain arithmetic operations, like multiplication and addition, using single logic gates, and is highly error-tolerant [101]. However, the compact arithmetic offered by SC comes at the cost of random errors and precision issues, which make it unsuitable for applications requiring high numerical precision[102]. Because of that, SC found its niche in error-tolerant applications, like image processing [102] and Discrete Fourier Transform [103]. Fortunately, neural networks' error-tolerant nature, as well as heavy reliance on linear algebra kernels and multiply-accumulate (MAC) operations, make them perfect candidates for SC acceleration [5, 64, 104]. Because of that, SC has been enjoying a renaissance, with many different flavors

of computation proposed, spanning various points on the accuracy-efficiency spectrum - some of them maximizing the density and efficiency, while others maintaining the accuracy close to fixed-point designs [5, 10, 64, 105, 104]. Below we briefly outline the most relevant components of SC.

1.2.1 Number Representation

Stochastic computing offers two alternative number representation formats: unipolar and bipolar. In the former, a value in the range of $[0, 1)$ is represented as the proportion of 1's in a binary stream of arbitrary length, as shown in Figure 1.1a). The latter, shown in Figure 1.1b), represents a number in the range of $(-1, 1)$ using the difference between the number of 1's and 0's in the stream. The streams are generated by comparing the possibility with a uniformly-distributed random number in $[0, 1)$. The stream generation circuits are commonly referred to as stochastic number generators (SNGs), consisting of a pseudo-random number source, e.g., a linear feedback shift register (LFSR) and a comparator [100], as shown in Figure 1.1c). Converting back to the binary domain can be done using a counter for unipolar representation.

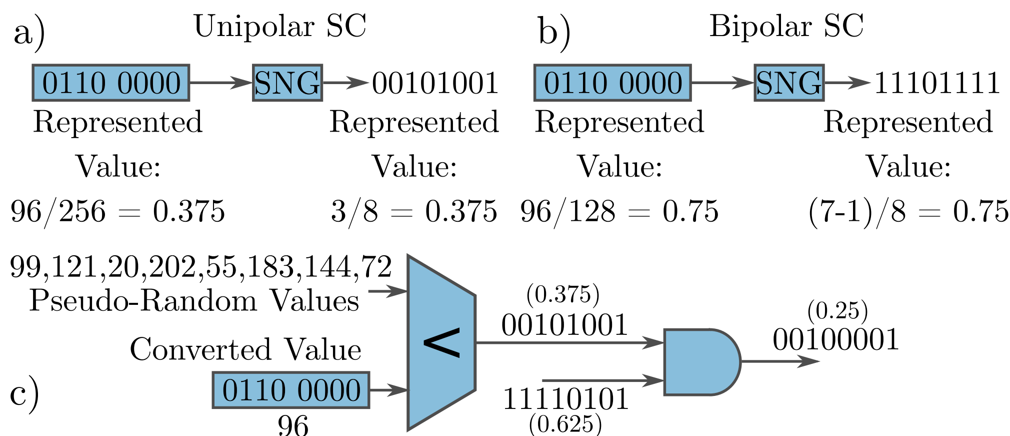


Figure 1.1: Examples of unipolar (a) and bipolar (b) SC representation, and a unipolar Stochastic Number Generator Circuit (SNG) with an AND gate SC multiplication (c).

In neural networks, maintaining high accuracy mandates using weights with both positive and negative values, which makes bipolar representation historically the most common choice when implementing SC-based accelerators [106, 107, 105]. However, [108] noticed that using unipolar representation results in higher accuracy compared with bipolar. Higher precision comes from the fact that the value of a unipolar bitstream has equivalent distribution to a binomial distribution divided by the length of the bitstream, so the root mean square (RMS) error can be represented as:

$$E_u = \frac{\sqrt{n_u v(1-v)}}{n_u} = \sqrt{\frac{v(1-v)}{n_u}} \tag{1.1}$$

Where E_u is the error, and n_u is the length of the unipolar bitstream. Error of bipolar stream can be calculated similarly as:

$$E_b = \sqrt{\frac{1-v^2}{n_b}} \tag{1.2}$$

Where E_b is the error, and n_b is the bitstream length. To have the same RMS error for both representations, $n_b = n_u \times (1+v)/v$, and $(1+v)/v > 2$ for $0 < v < 1$. Consequently, the bipolar representation will always require $> 2X$ bitstream length to represent the same value with the same accuracy compared to unipolar representation. Figure 1.2a) shows the simulated error of software-generated stochastic numbers across 10,000 runs for different values, confirming the theoretical results. Because of that, recent works have embraced unipolar representation with extensions to make using negative numbers possible [109, 110].

1.2.2 Multiplication and Accumulation

One of the main selling points of SC is that it can perform computation using bit-wise operations between two input bitstreams. To explain the reason behind that, consider stochastic numbers as probabilities. For example, an AND gate performs $AND(v_1, v_2) = v_1 \times v_2$, where v_1, v_2 are the input possibilities for two unipolar streams. Similarly, a 2:1 multiplexer

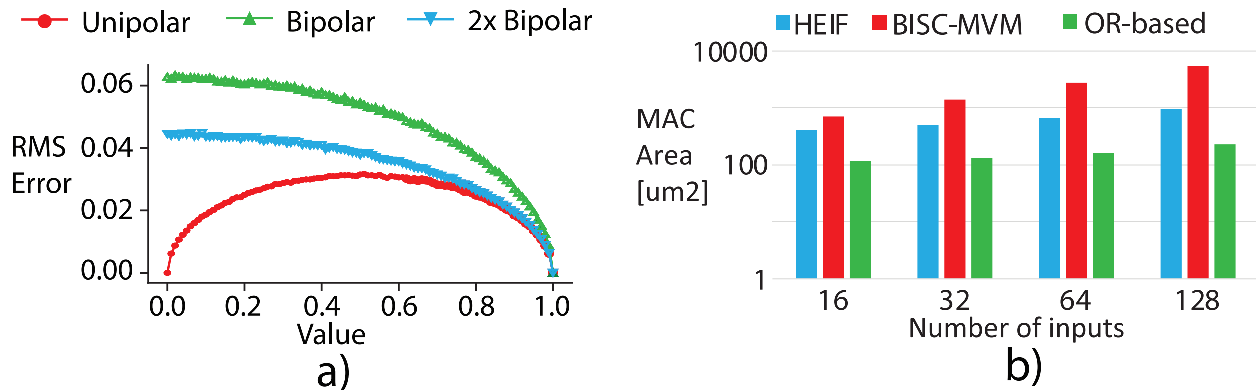


Figure 1.2: RMS error of the bit stream representation for a given value a), and area [μm^2] comparison between different SC MACs, depending on the number of inputs b).

(MUX) can be used to compute a scaled addition between two inputs: $MUX(v_1, v_2, s) = s \times v_1 + (1 - s) \times v_2$, where s is the select input. For a bit-wise computation, each output bit follows a Bernoulli distribution with a mean equal to the expected value if input bits are independent. As a result, the output accuracy of bit-wise computation has the same accuracy as generating the same number directly. This observation allows the modeling of output error using only the expected output value without worrying about input values and computation performed.

Accumulation has historically been an issue when applying stochastic computing to neural networks, mainly due to precision [106]. Multiplexer can act as a stochastic adder by using a 50% random stream at the select input. However, multiplexer-based addition suffers from two critical issues: it requires the generation of an additional random stream and scales down the result by two compared to typical addition. Sharing a random sequence can alleviate the former problem. The latter issue degrades the accuracy of computation, mainly when performing extensive accumulation. Since neural networks generally perform very large matrix multiplications, fully stochastic MUX-based addition is not suitable for those without a significant increase in the stream length [106]. Because of that, prior works in SC-based

neural network acceleration were often forced to perform accumulation in the fixed-point domain, by either using costly approximate parallel counters (APCs) [106] or converting the results every multiplication [107].

An alternative way of stochastic accumulation, OR-based accumulation, has been proposed in [111]. It is scaling-free (important for extensive accumulations in DNNs), and also much more compact than alternative accumulation methods. However, it has reasonable accuracy only for unipolar streams. This property has made it largely disregarded in prior SC work [106]. Figure 1.2b) shows the comparison of the MAC area for two prior state-of-the-art methods, SC-DCNN/HEIF [106, 105] and BISC-MVM [107] to OR-based MAC for a different number of inputs synthesized using a commercial 28nm library. SC-DCNN/HEIF uses APC-based accumulation, while BISC-MVM converts results to fixed-point binary after every multiplication. As can be seen, OR accumulation can be as much as 4.2x and 23.8x more compact than SC-DCNN/HEIF and BISC-MVM, respectively. As we will show in Section 3.3.1, taking advantage of computational density is vital to reap the full benefits offered by SC, which is why using OR-based accumulation is so important. However, a major issue with it is that it is not an exact addition. For a two-input OR, the result is equal to $v1+v2-v1v2$ instead of $v1+v2$. We show later how we address this imperfect accumulation in the training of the networks.

1.2.3 Stochastic Neural Network Functions

More complicated operations, like neural network activation functions, are often implemented using finite-state machine (FSM) circuitry [106, 105]. Many recent efforts have been directed towards finding optimal SC representations for particular operations [112, 113, 114, 115]. Another essential operation is max pooling, which is the most commonly used method of dimensionality reduction in neural networks [27, 1]. Although prior works have shown successful implementation of max pooling in SC [106, 108, 116], it usually requires a finite state machine (FSM), which increases the area and energy cost of SC. In contrast, to implement

average pooling, only a single multiplexer exploiting the scaled addition property is required. For example, [116] have shown that replacing max with average pooling, can reduce the area and power of a stochastic convolutional engine by up to 2.02x and 1.94x respectively.

1.3 Domain-Specific Acceleration - Breaking the Shackles of Generality

Even apart from the emergence of deep learning, the need for domain-specific (DS) computing has been widely recognized [12, 117]. However, the rapid popularization of first vision [27, 1, 30], and then language models [118, 119], made the adoption of DS accelerators a priority. Initial approaches, like Eyeriss [2] and DianNao [120], focused on reducing the precision of computation from floating- to fixed-point, and optimizing dataflows targeted at convolutional neural networks (CNNs). From there on, multiple different approaches followed. Some focused on finding new hardware opportunities in quantization, in particular using bit-serial arithmetic, which serially processes fixed-point numbers, one bit position at a time. Examples of such accelerators include Stripes [19], UNPU [121], Laconic [122], and Pragmatic [123], and Bit Fusion [124]. There are also binary and ternary hardware accelerators, listed in Section 1.1.1. Others have tried designing architectures oriented around exploiting sparsity, like SCNN [79], EIE [81], NullHop [125], GoSPA [126], SparTen [127], and Tensaurus [128], among others. A more comprehensive discussion of sparse accelerators is presented in Chapter 5. Extensive work has also been done on optimizing dataflows [129, 130], and scale-out architectures [15, 131].

Given the limits of efficiency within the digital domain, and highly regular, dot-product-based arithmetic involved in neural networks, people quickly recognized the potential of analog computation. Neuromorphic approaches have already been discussed in Section 1.2.1. However, they were quickly overshadowed by non-spiking analog accelerators based on the idea of compute-in-memory (CIM) [132]. They combine the advantage of the advantage of

dense and efficient analog computation, with reduced data movement, as one of the operands does not need to leave the memory. Different types of memory cells can be used, providing different benefits [132, 133, 56, 134, 135, 136, 137, 138]. This is an extensive field, and while more details are provided in Chapter 6, interested readers should refer to [25].

1.4 Dissertation Outline

The research in this dissertation was motivated by a single goal: enabling ever-larger deep learning models to run on even the most resource-constrained edge devices. This goal is achieved by identifying the most promising techniques, coming up with novel solutions that address their shortcomings, combining them in a synergistic manner, and creating efficient highly hardware and software implementations.

This dissertation is organized as follows:

- **Chapter 2** describes *3PXNet - Pruned, Permuted, Packed XNOR Networks*, which combine binarization and pruning to deliver usable neural network models deployable on the most tightly constrained platforms.

- **Chapter 3** proposes *ACOUSTIC - Accelerating Convolutions through OR-Unipolar Skipped Stochastic Computing*, the first system-level accelerator architecture that attempts to maximize the synergies of deep convolutional neural networks and stochastic computing. A 14nm demonstration chip is also demonstrated.

- **Chapter 4** presents *GEO - Generation and Execution Optimized Stochastic Computing Accelerator for Neural Networks*. GEO addresses the main shortcomings of ACOUSTIC, through novel algorithmic and architectural solutions.

- **Chapter 5** describes *SASCHA – Sparsity-Aware Stochastic Computing Hardware Architecture for Neural Network Acceleration* a first accelerator that combines both

stochastic computing and weight sparsity support by solving a series of non-trivial implementation challenges to enable a new frontier in energy-efficiency.

- **Chapter 6** presents SCIMITAR - an event-based, stochastic compute in-memory accelerator architecture. It synergistically merges SC with the emerging field of in-memory computation, while enabling support for highly-sparse event-based data processing.

- Finally, **Chapter 7** summarizes the contributions of this dissertation, and describes its possible future extensions.

CHAPTER 2

3PXNet - Fewer Bits, More Zeros

As the adoption of Neural Networks continues to proliferate different classes of applications and systems, microcontroller-based edge devices have lingered behind. Their strict energy and storage limitations make them unable to cope with the sizes of popular neural network models. While many compression methods, such as precision reduction and pruning, have been proposed to alleviate this, they do not go quite far enough. To push size reduction to its absolute limits, we combine binarization with sparsity in *Pruned-Permuted-Packed XNOR Networks (3PXNet)*, which can be efficiently implemented on even the smallest of embedded microcontrollers. 3PXNets can reduce model sizes by up to 38X and runtime by up to 3X compared with already compact conventional binarized implementations with less than 3% accuracy reduction. We have created the first software implementation of sparse-binarized Neural Networks, released as an open-source library targeting edge devices. Our library is complete with training methodology and model generating scripts, making it easy and fast to deploy.

Collaborators:

- Tianmu Li, Electrical and Computer Engineering, UCLA.
- Professor Puneet Gupta, Electrical and Computer Engineering, UCLA.

Source code available at: `https://github.com/nanocad-lab/3pxnet/`

2.1 A Case for Sparse XNOR Networks

Given the significant benefits of binarization and sparsity on their own, it is not immediately obvious why we would want to combine them. However, consider the available memory in some of the common embedded microcontroller platforms, shown in the first part of Table 2.1. Their storage is usually limited to few hundred kilobytes at most. Such severe resource constraints make implementation of reasonable deep learning networks on these platforms challenging. For example, consider few network models shown in Table 2.2. Most floating point and even 8-bit fixed point implementations are 10-1000X off from where they need to be for these platforms.

Table 2.1: Hardware platforms used for the runtime experiments. Only the *Large* platform has a DSP extension with hardware multiply-accumulate unit. All three microcontrollers are from the ST Nucleo family [139].

Name	Model	SRAM (KB)	Flash (KB)	Core Type	Clock (MHz)
NUC Large	F746ZG[139]	320	1024	ARM CM7	216
NUC Medium	F103RB[139]	20	128	ARM CM3	72
NUC Small	F031K6[139]	4	32	ARM CM0	48
Name	Model	L2 (MB)	DRAM (GB)	Core Type	Clock (GHz)
Raspberry Pi	Model B+ [140]	2	1	ARM CA53	1.2

By constraining weights and activations to binary values, Binarized Neural Networks can perform 32 multiply-accumulate operations using XNOR and population count (popcount) instructions in a 32-bit processor (with appropriate "*packing*" of weights and activations),

which gives it a potential 32x storage and computation saving compared to a 32-bit implementation (see Table 2.2). While this in itself is impressive, it might not be enough to use popular models on typical embedded development platforms. Further compression is therefore necessary to use large models on those devices, or in case of the most memory-constrained ones, make it feasible to deploy them at all.

Table 2.2: Weight storage requirements of different networks depending on precision.

Network	Weight Memory [MB]		
	F32	FP8	XNOR
ILSVRC VGG-D [1]	553.4	138.3	17.3
ILSVRC AlexNet [27]	227.5	56.9	7.1
MNIST MLP [141]	147.2	36.8	4.6
MNIST MLP Small (This work)	0.40	0.10	0.01
CIFAR-10 CNN [141]	56.1	14	1.7
CIFAR-10 CNN Small [95]	0.36	0.09	0.01

In this Chapter, we propose a Pruned, Permuted, and Packed XNOR Neural Network (3PXNet) model aiming to combine binarization and pruning in a way that is computationally efficient and does not significantly degrade accuracy. We specifically target resource-constrained edge devices and provide implementation results on a range of embedded platforms. Our pruning method allows further model size reduction and speedup compared to binarized networks. Contributions of this work are as follows:

- We develop methods to prune binarized XNOR networks, aware of the need for packing them into words for computational efficiency.
- We develop training methods for such 3PXNets and open-source the training routines using PyTorch framework [142].

- We show that 3PXNets offer some of the most compact networks with good accuracy: 3x-38x (22x-307x) size reduction versus dense binary (8-bit), with 0-5.2% (0.3-10.4%) accuracy drop on MNIST and 2.3-3.8% (3.5-5%) on Google Speech dataset, depending on the level of sparsity.
- We develop the *first* software implementation of sparse binarized networks and open-source implementation of 3PXNets.
- We make multiple design optimizations, like loop ordering, fused kernels, and implicit padding, which result in a compact memory footprint and runtime. 3PXNet implementation can be as much as 3X (25X) faster and more energy efficient than dense binarized (8-bit fixed point) networks enabling real-time inference on IoT platforms.

2.2 The 3PXNet Approach

In the following sections we describe the principal components of the Pruned-Permuted-Packed XNOR Networks.

2.2.1 Challenges in pruning XNOR networks

Binarized neural networks reduce network size by having only one bit for each weight, allowing packing of multiple weights in a binary vector, e.g., a processor 32-bit word. By constraining activations to binary values, they can also be packed, and 32 multiply-accumulate operations (MAC) can be performed in parallel using a bitwise XNOR and population count (popcount, or Hamming weight) instructions on the activation and weight packs in a 32-bit processor (or 64 MACs in a 64-bit processor). In theory, this can reduce weight storage and computation requirement by a factor of 32 compared to a 32-bit floating point implementation.

Dense binarized or XNOR networks are relatively straightforward to pack for both weights

and activations, thereby getting close to the theoretical 32X improvement [46]. However, introducing sparsity on top of binarization will not necessarily improve the results further. We first illustrate how naively pruning binarized networks actually worsens storage and runtime. Consider a "small" binarized MLP used for MNIST (see Table 2.2) classification with the input layer of size 784, one hidden layer with 128 neurons, and an output layer of size 10, both followed by batch normalization [143]. If we prune it without any constraints and store the sparse weight matrix using compressed-sparse-row (CSR) format [144], binary weight packing and "SIMD" XNOR multiplication cannot be easily leveraged. We refer to this scheme as *Naively-Pruned (NP)* network. If the number of non-zero weights for each kernel is constrained to be the same multiple of 32, binary weights can now be packed to save storage, but activations need to be fetched individually and packed separately for each kernel during computation. We refer to this scheme as *Naively-Pruned, Packed Network (NPP)*. NPP scheme has two advantages over NP in terms of storage overhead. First is packing weights into binary vectors instead of storing values individually. Second is getting rid of row extent values in the CSR format - having the same number of packs per kernel means that only one row extent value per layer needs to be stored. This benefit will have a more profound impact at high sparsity levels, because while the number of column indices goes down with sparsity, the number of rows, and therefore row extents, stays the same. Figure 2.1 shows the total storage required for NP and NPP implementations, compared to a dense one. NPP offers significant storage reduction over NP, mainly through a reduction in weight storage itself. However, to break even with Dense XNOR, sparsity levels of over 90% and 95% are required for NPP and NP, respectively.

While the NPP scheme allows for packing non-zero weights, there is no easy way to leverage input packing. As runtime is usually a concern for convolutional layers, we implemented both dense and NPP kernels for the Large CNN model (Table 2.3) for CIFAR-10. Even with sparsity set to 87.5% for each convolutional layer, except for the first one, which is kept as dense binary-weight (BWN - using full-precision activations and binarized weights

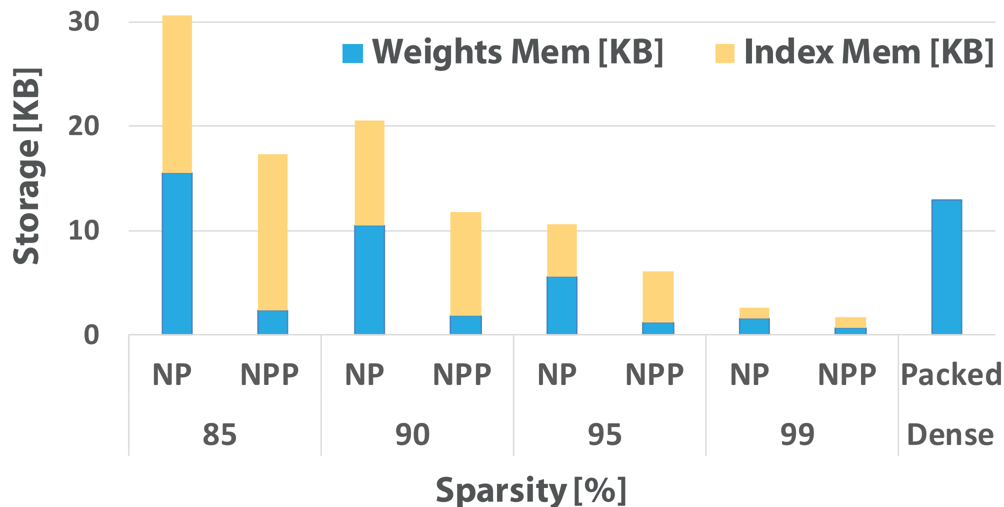


Figure 2.1: Storage requirements for Dense, NP and NPP XNOR "small" MNIST MLP for varying levels of sparsity.

[141]), NPP version is 15X-29X slower compared to the unpruned dense XNOR Net on the different convolutional layers, running on a Raspberry Pi 3. This result clearly shows that naively pruning binarized networks, even with packing, is not beneficial at best and possibly detrimental to both their size and runtime.

2.2.2 Pruning a packed XNOR network

To make inference of pruned binarized neural networks more efficient, inputs (activations) need to be fetched in packs, and those packs need to work for all kernels in a layer. This requirement leads us to constrain the non-zero, or "active", weights of each kernel to packs of 32 consecutive positions. These 32-bit packs are aligned across all kernels of a layer so that the activations only need to be packed once. If two kernels have active packs with the same index, they will be multiplied with the same activation pack. This packing constraint reduces the number of indices to store by a factor of 32 compared to NPP implementation.

Forcing the packing constraint reduces the flexibility of the network and can result in excessive pruning in some packs and insufficient pruning in others. To alleviate this effect,

Table 2.3: Benchmark models and datasets

Dataset	Model		Architecture
MNIST	MLP	Large [141]	784-4096-4096-4096-10
		Small	784-128-10
	CNN	Small	$32CONV5 - MP2$ $32CONV5 - MP2$ $10FC$
CIFAR-10 SVHN Speech[145]	CNN	Large [141] Medium	$128CONV3 \times 2 - MP2$ $256CONV3 \times 2 - MP2$ $512CONV3 \times 2 - MP2$ $(1024FC \times 2) - 10FC$

we propose to permute the weight matrix so that weights inside the 32-bit packs are more likely to be all zeros in the 3PXNet network. Figure 2.2 illustrates the effect of permutation on packing for a pack size of 4. A similar approach appears in [91] albeit in the context of crossbar neuromorphic systems. Weight permutations are performed on input channels (NI) of a fully-connected or convolutional layer. For a convolutional layer with weight shape (KN, KZ, KY, KX), the weight is first flattened in all dimensions except KZ to (KN_flat, KZ), and then treated as a fully-connected layer.

Permutation tries to group similar input channels into packs of 32 in a method resembling Prim's algorithm [146]. Before grouping, weights are first ternarized to {-1,0,+1}. For each pack, a random input channel is chosen as starting point. A similarity score is calculated by counting the overlap of 0 positions between the existing input channels in the pack and all other channels that have not been grouped, and the channel with the most overlap is added to the pack. If a group has both 0s and {-1, +1} values in a position, it is considered as a non-zero position, as it's unclear if weights in the pack will be pruned or not, and either

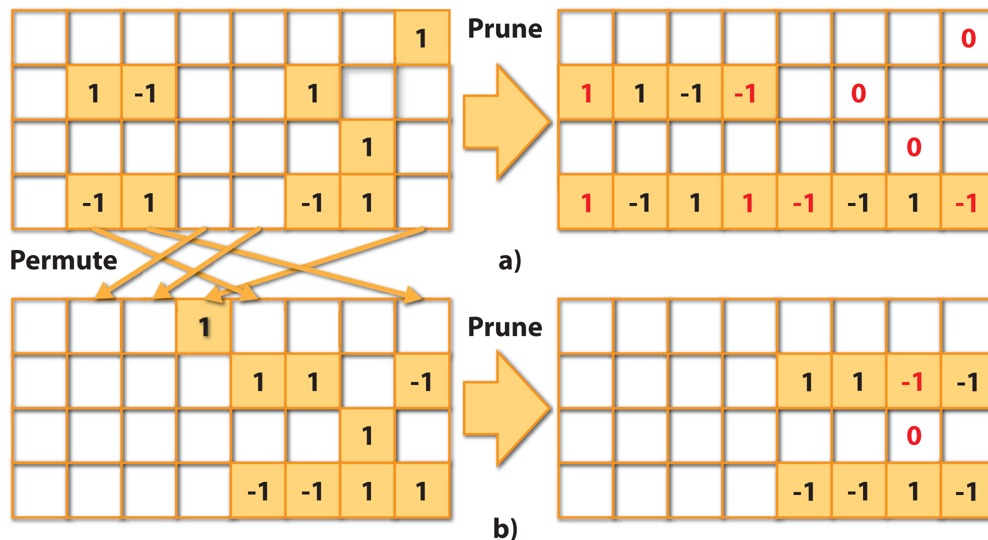


Figure 2.2: Pruning with packing constraint of 4 bits a) without permutation, b) with permutation

choice will result in some weights being forced to change. On the other hand, positions filled with 0s in a pack will definitely be pruned, and the pruning action will not force any weight change. Once a pack is filled, another random input channel is chosen as the starting position for the next pack. This process continues until all input channels are grouped into packs. Input channel permutations can be directly translated to output channel reordering of the previous layer, so it is completely free in terms of inference except for the first layer. Figure 2.3 shows the effect of using permutation in a small MLP, where maximal benefit ($> 4\%$) is observed with very high sparsity.

2.2.3 Training 3PXNets

Network pruning is usually performed by training a dense network and then deleting some edges or kernels [22, 80]. Since 0 cannot be represented in a binarized XNOR network, and the binary values contain no information about the importance of each weight, we start the training with ternary weights to introduce zero values. The training algorithm is adapted

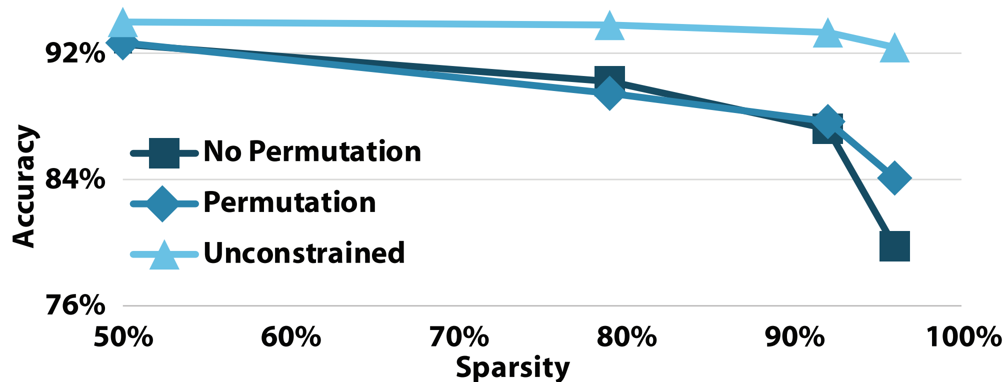


Figure 2.3: Comparison of training results with and without permutation for different sparsities.

from the one in [46] where full-precision weights are kept during training, and are binarized during inference, except that the weights are ternarized using a threshold function instead of binarization.

```
weight_sorted = sort(weight, descending=True)
index = ceil((1-sparsity)*size/word_width) * word_width
thres = weight_sorted[index-1]
```

Algorithm 2.1: Pseudocode for calculating threshold for NPP.

The threshold used for pruning is calculated using Algorithm 2.1. The weight tensor is first flattened and sorted based on the floating point values. The threshold value is then chosen to make sure that the sparsity roughly aligns with the sparsity target of that layer, and that the number of active weights aligns with the word width of the processor (typically set to 32). The result of this phase of training follows the NPP scheme, which allows efficient storage of weights but requires high indexing overhead. We then permute the weight matrices using the method mentioned above and enforce the packing constraint.

```
weight_split = split(weight, split_size=word_width)
// For every split weight pack
for sp = 0 to size/word_width
```

```
4     weight_sum[sp] = sum(abs(weight_split[sp]))
5 weight_sorted = sort(weight_sum, descending=True)
6 index = ceil((1-sparsity) * size / word_width)
7 thres = weight_sorted[index-1]
```

Algorithm 2.2: Pseudocode for calculating threshold for packed pruning.

Similar to the NPP phase, a threshold is calculated using Algorithm 2.2 to determine the packs to prune. For each kernel in the weight matrix, we split the weight values into packs aligned to the word width of the processor, and sum the absolute value of weights inside each pack. We then calculate the threshold so that the sparsity roughly matches the sparsity requirement of the layer, and prune away the packs with sums below the threshold. For packs that are not pruned, the weights inside are forced into {-1, +1}, even if they were originally 0. Before pruning is fixed, the pruned packs can still be unpruned if the sum of another pack drops lower than the pruned pack. The network is trained for a few more epochs to determine which packs to prune. Finally, the packs to be pruned are fixed, and the model is fine-tuned to further improve accuracy. The final pruned, permuted and packed binarized network is referred to as 3PXNet.

As shown in Figure 2.3, forcing the packing constraint reduces network accuracy compared to unconstrained pruning, and permutation cannot fully recover accuracy loss. Permutation of inputs changes the allowed topology of the final network, which is the case for MLPs. For CNNs we are not pruning the input layer, so permutation does not change the achievable topology. Due to the fact that we are permuting entire kernel planes, permutation is also more restrictive for convolutional layers. Immediately after permuting, packing and pruning a trained network, permutation has a significant advantage in accuracy (as much as 20% from 23.0 to 43.2%) but we fine-tune the network after permutation and it largely recovers irrespective of permutation indicating that the networks have significant redundancy unless the sparsity is very high ($> 90\%$). Because of the negligible effect of permutation on

training and inference runtime, all networks are trained with and without permutation, and the better performing one is chosen as result.

2.3 Implementing 3PXNet

In this Section, we detail the implementation choices which make 3PXNet one of the most compact and efficient neural network compression schemes. Before doing that however, we want to establish a consistent terminology for describing both fully-connected (FC) and convolutional (CN) layers. We will be using uppercase variables to describe dimensions, and lowercase ones to describe indices within those dimensions. Fully-connected layer is essentially a matrix-vector product between a vector of NI activations, or inputs, and a matrix of weights with size $NO \times NI$, producing NO outputs, as shown in Figure 2.4. Number of outputs is the same as the number of kernels, and each kernel is a row in the weight matrix, with a size equal to the number of inputs.

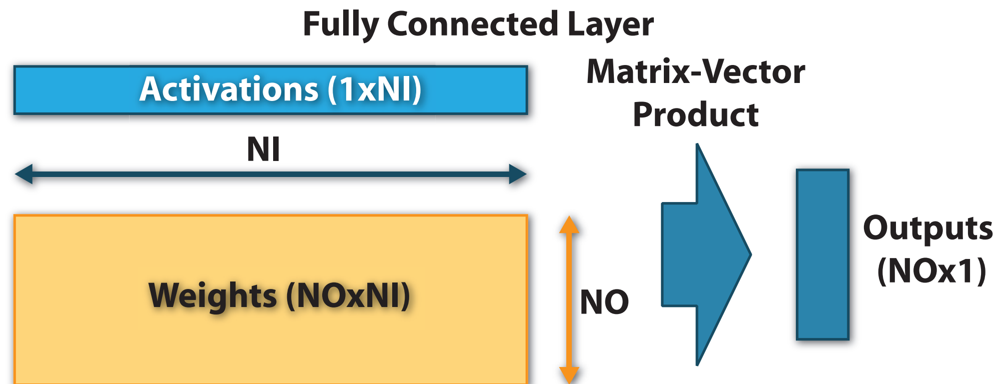


Figure 2.4: A schematic view of a fully-connected layer with NI inputs and NO kernels.

A convolutional layer is shown schematically in Figure 2.5. Input to the layer is a 3D activation tensor of height Y, width X and depth Z. Activations are often padded in the X and Y dimensions, creating a "halo". Padding size, PD, is applied on both sides of each dimension, creating a tensor of height $Y+2PD$, width $X+2PD$, and depth Z. This tensor is

then convolved with KN kernels, each with height KY, width KX and depth KZ. We only consider cases where input (Z) and kernel (KZ) depths are the same [1, 30]. Each kernel generates an output of height $Y+2PD-KY+1$, width $X+2PD-KX+1$, and depth of 1. Those outputs are then "stacked" depth-wise, creating a tensor of height $Y+2PD-KY+1$, width $X+2PD-KX+1$, and depth of KN. Certain convolutional layers will be followed by pooling operations to reduce output dimensionality. Pooling, most commonly max pooling, uses a kernel with size and stride PL in the Y, and X dimensions, producing an output of height $OY = (Y+2PD-KY)/PL+1$, width $OX = (X+2PD-KX)/PL+1$, and depth KN. Those equations can be used for any type of layer - if padding is not used, PD is set to 0, and if there is no pooling, PL is set to 1. For pooling operations, kernel size and stride might sometimes be different, but that is not the case for any of the networks implemented here [27]. For simplicity we also do not discuss strided convolutions [30].

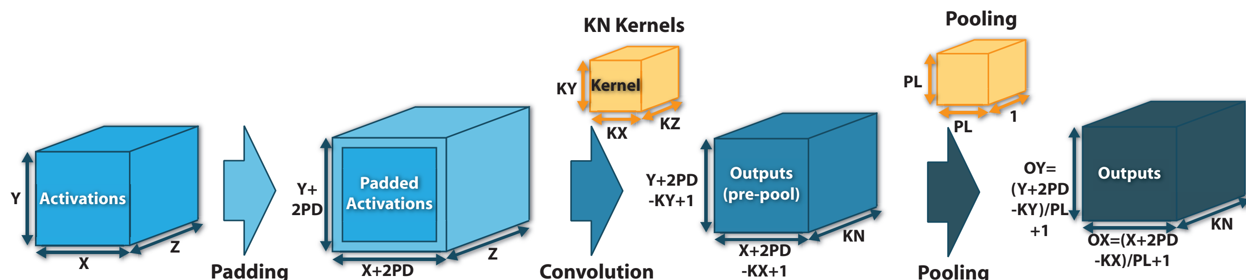


Figure 2.5: A schematic view of a padded convolutional layer followed by pooling operation.

2.3.1 Fully-Connected Layers

As mentioned above, a fully-connected layer is essentially a general matrix-vector multiplication (GEMV) kernel. As it is easy to obtain close to theoretical speedups even with a straightforward GEMV implementation, we use it for our dense binarized reference [59]. Activations and weights are packed into binarized vectors which size is a multiple of 32 or 64, depending on the bitwidth of the processor used. In all subsequent sections we assume

a 32-bit pack width. The exact alignment to the multiple of 32 is not necessary and could be handled by masking and adjusting the last pack to arbitrary size. However, for common network topologies, like the one presented in this work, layer sizes are generally a multiple of 32. We use the outputs/kernels as an outer loop, and input packs as an inner loop, as shown in Algorithm 2.3 for 32-bit packs. After all popcounts for a given output are completed, the result needs to be adjusted. This is because the popcount result is the number of ones, whereas the actual result is the number of ones minus the number of zeroes, since zeroes represent -1. The advantage of using outputs as an outer loop is that only one output is accumulated to at a time, and its partial result can be kept locally. On the other hand, it can make input reuse in caches harder. However, binarized input vectors are usually in the order of few hundred bytes, meaning even the smallest caches can fully fit them. After a single output is computed, we then perform on-the-fly binarization, to pack outputs into vectors for the next layer, as shown in Algorithm 2.3.

```
1  # For every output pack
2  for no = 0 to NO/32-1
3     # For every output in a pack
4     oPack = 0
5     for pCnt = 0 to 32-1
6        output = 0
7        # For every input
8        for ni = 0 to NI/32-1
9           # XNOR multiplication
10          mult = xnor(inputs[ni],weights[no*NI+ni])
11          # Popcount accumulation
12          output += popcount(mult)
13       # Correct output value
14       output = output-(NI*32-output)
15       # Binarize
16       output = output >= 0
17       # Shift and pack
```

```
18       oPack |= output << (31-pCnt)
19    outputs[no] = oPack
```

Algorithm 2.3: Pseudocode showing dense FC layer processing.

For 3PXNets, we only store active (non-zero) weights. We constrain pruning such that every kernel has the same number of non-zero weight packs, NP. We did not enforce this constraint initially, but we found that introducing it does not hurt accuracy. Models trained with the same size of sparse kernels sizes never have more than 0.5% lower accuracy than models without that constraint. Additionally, it enables a further reduction in indexing overhead, by not storing row extents of the CSR format. For example, for MLP-L with high sparsity we can achieve up to 50% reduction in indexing storage and 16.7% overall. We call those active packs. Because we know the number of packs for each neuron, only one index per pack needs to be stored, making it more efficient than e.g., the CSR format. We use 8-bit indices, which support layers of up to 8192 activations. A simplified representation of weight and index storage for 3PXNet FC layers is shown in Figure 2.6. The loop ordering is the same as for dense implementation, but the inner loop iterates only through the active packs instead of all inputs, as shown in Algorithm 2.4. Both the dense and 3PXNet FC layer implementations have versions with and without output binarization, the latter used for the final classifier layer.

```
1  # For every output pack
2  for no = 0 to NO/32-1
3     oPack = 0
4     # For every output in a pack
5     for pCnt = 0 to 32-1
6        output = 0
7        # For all active packs
8        for np = 0 to NP-1
9           # Fetch correct input pack
10          input = inputs[indcs[no*NP+np]]
```

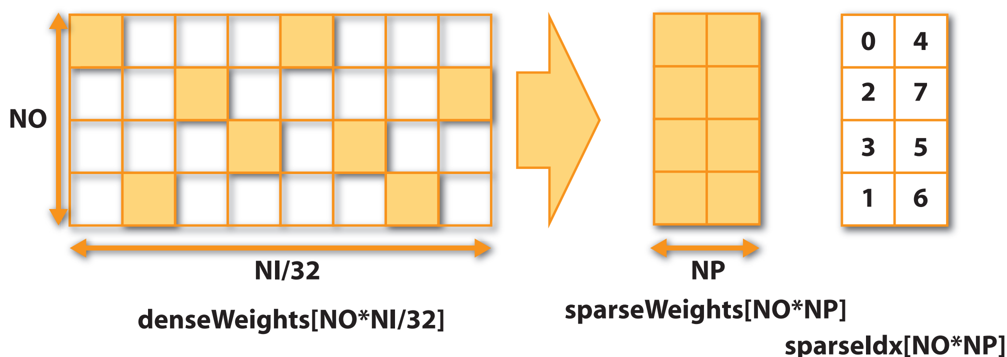


Figure 2.6: 256x4 fully-connected layer weight and index storage with 75% sparsity (NP=2x 32-bit packs per output).

```
11            # XNOR multiplication
12            mult = xnor(input,weights[no*NP+np])
13            # Popcount accumulation
14            output += popcount(mult)
15        # Correct output value
16        output = output-(NP*32-output)
17        # Rest as in Alg 2.1
```

Algorithm 2.4: Pseudocode showing 3PXNet fully-connected layer processing.

2.3.2 Convolutional Layers

Convolutional layers can be unrolled into matrix-matrix multiplication, as proposed by Chellapilla et al. [147], which makes it possible to compute them in the same way as FC layers. However, the overhead of unrolling into matrix form in binarized implementations offsets the arithmetic speedup [61]. To address that, we implement dense convolutional layers directly using the PressedConv approach proposed by Hu et al.[61]. We capitalize on the fact that most convolutional layer activations have a depth (Z) which is a multiple of 32 and organize our data using depth as the innermost dimension for both activations and kernels. Because

of that, we can easily pack the activations and kernel weights in vectors of 32, irrespective of X and Y dimensions, which can have arbitrary values depending on the network. This is schematically shown, for a $Y = 3, X = 3, Z = 32$ kernel, in Figure 2.7. When implementing convolutional layers, loop ordering plays a particularly important role. There are at minimum 6 loops: output height (OY), output width (OX), output depth (KN), kernel height (KY), kernel width (KX), and kernel depth (KZ). For dense XNOR implementation, we use $OY-OX-KN-KY-KX-KZ$ ordering, as shown in Algorithm 2.5. The kernel (KN) loop is split into two loops to facilitate packing, similar to the output loop in fully-connected layers. The advantage of this ordering is that it provides good locality on both activations and outputs. Figure 2.8 shows a comparison of speedups for a few different VGG-16D [1] for dense and 3PXNet with the kernel as an outer ($K - YX$) and inner ($YX - K$) loop, normalized to dense $K - YX$. Using kernel as an inner dimension results in 8% geomean speedup.

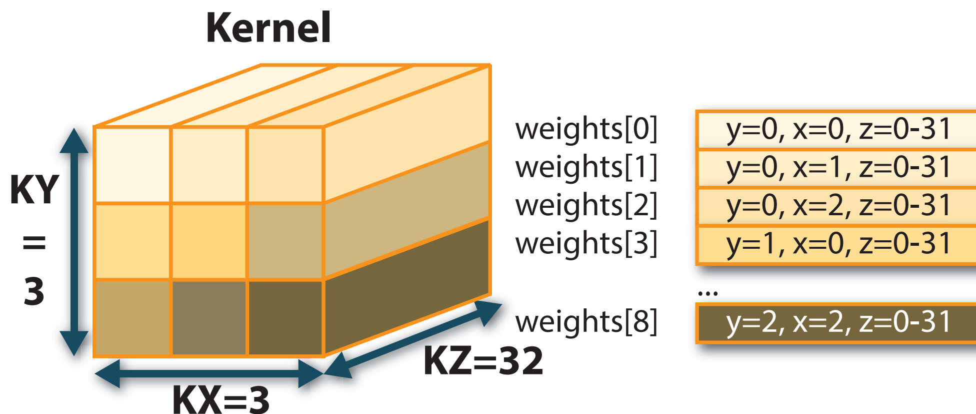


Figure 2.7: 3x3x32 kernel packed into depth-first binarized vectors.

```
1 # For every output row
2 for oy = 0 to OY-1
3    # For every output column
4    for ox = 0 to OX-1
5       # For every kernel packet
6       for kn = 0 to (KN/32)-1
```

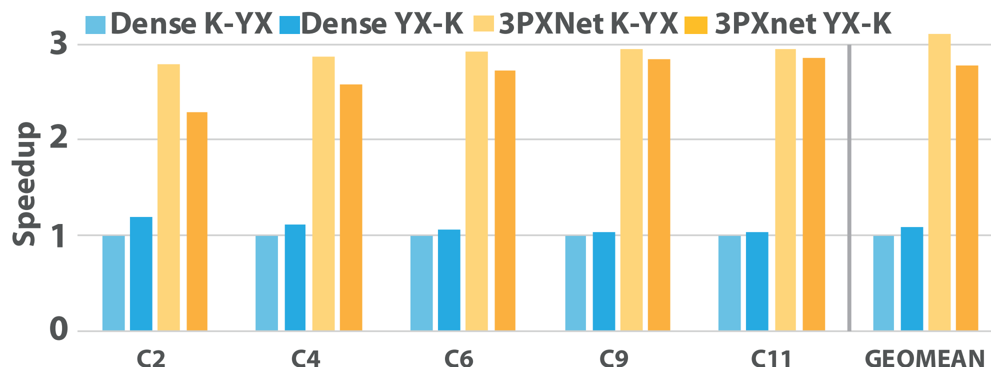


Figure 2.8: Dense and 3PXNet (93.75% sparsity) speedups for kernel as an outer (K-YX) and inner (YX-K) loop for different VGG-16D [1] layers, normalized to dense K-YX.

```
7              # For every kernel in a packet
8              for ks = 0 to 32-1
9                 # For every kernel row
10                for ky = 0 to KY-1
11                   # For every kernel column
12                   for kx = 0 to KX-1
13                      # For every pack
14                      for kz = 0 to KZ/32-1
15                         # XNOR multiplication
16                         # Popcount accumulation
17                # Output correction
18                # Packing
```

Algorithm 2.5: Pseudocode showing dense conv layer loop ordering.

Similarly to FC layers, for 3PXNets, we only store the non-zero weight packs and their corresponding indices. We also constrain each kernel to have the same number (KL) of active packs to simplify indexing. This is shown schematically in Figure 2.9, for a kernel with $KY = 3$, $KX = 3$, $KZ = 32$, and $KL = 3$ active packs. We only use one index per pack, which combines information on all three dimensions, compressing index storage by a

factor of 3. It comes at a cost to runtime, as indices need to be decoded for every kernel.

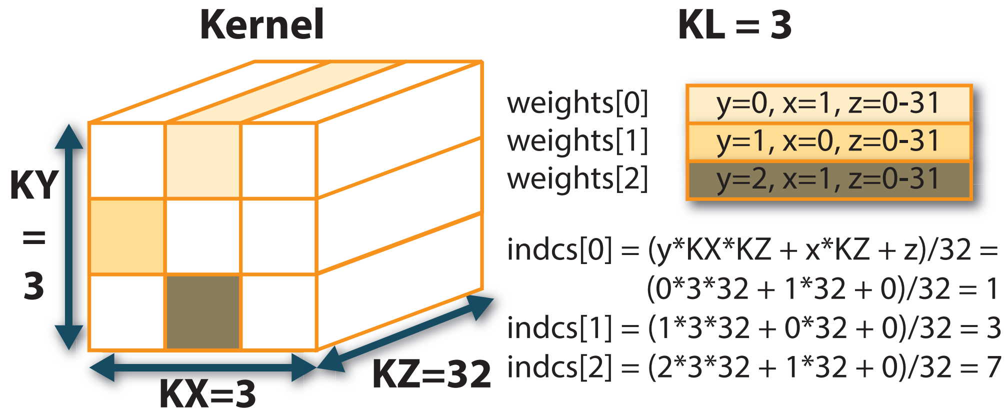


Figure 2.9: 3x3x32 Convolution kernel weight packs and indices with KL=3 active packs.

For 3PXNet convolutional layers, we change the loop order compared to the dense implementation. As shown in Figure 2.6, for 3PXNet, using kernel as an outer dimension is faster. This is because it amortizes the cost of fetching and decoding indices - this way it only needs to be done once per kernel. The downside of this approach is reduced output locality. Therefore we use kernels (KN) as an outer loop, as shown in Algorithm 2.6.

```
1  # For every kernel packet
2  for k = 0 to (KN/32)-1
3     # For every kernel in a packet
4     for ks = 0 to 32-1
5        # Decode indices
6        # For every output row
7        for oy = 0 to OY-1
8           # For every output column
9           for ox = 0 to OX-1
10             # For every active pack
11             for kl = 0 to KL-1
12                 # XNOR multiplication
13                 # Popcount accumulation
14             # Output correction
```

```
15              # Packing
```

Algorithm 2.6: Pseudocode showing 3PXNet conv layer loop ordering.

For padding support, to further reduce storage requirements for intermediate activations, we opt against storing padded regions explicitly in memory, as they don't contribute to output values. We detect multiplications that fall under the padded regions and skip related computation. This approach is possible because convolution is done depth-first in packs of 32 or 64 values and each pack falls completely inside or outside the padded region. Skipping decision is therefore made on a single pack granularity. Other binarized works have proposed doing padding computation explicitly, through -1 or +1 padding, without significantly affecting accuracy [48, 50] albeit with increased storage and runtime. For example, using explicit padding in convolutional layers of CNN Large, listed in Table 2.3, adds between 12.8% and 56.3% additional activation storage and increases computation by 4.1% to 31.9%, depending on the layer. Because padding is only ever applied in X and Y dimensions and packing is done along the depth, every pack is either completely in the padding region or not. Since this approach makes computation irregular, potentially worsening runtime, we further split activations into five regions where the middle one can be computed without padding-related overheads, and the "halo" regions use computation skip, as shown in Figure 2.10.

2.3.3 Fused kernels

Because we are targeting resource-constrained embedded devices, we want to limit intermediate storage used during the computation. In XNOR Networks, outputs of a given fully-connected or convolutional layers are generally passed through pooling or batch normalization layers before they can be binarized [27, 143], meaning that intermediate results need to be stored in full precision. To alleviate this issue, we use fused kernels, similar to McDanel et al. [63]. Operations following fully-connected and convolutional layers, such as pooling and batch normalization, are performed on-the-fly for each output. Specifically, they

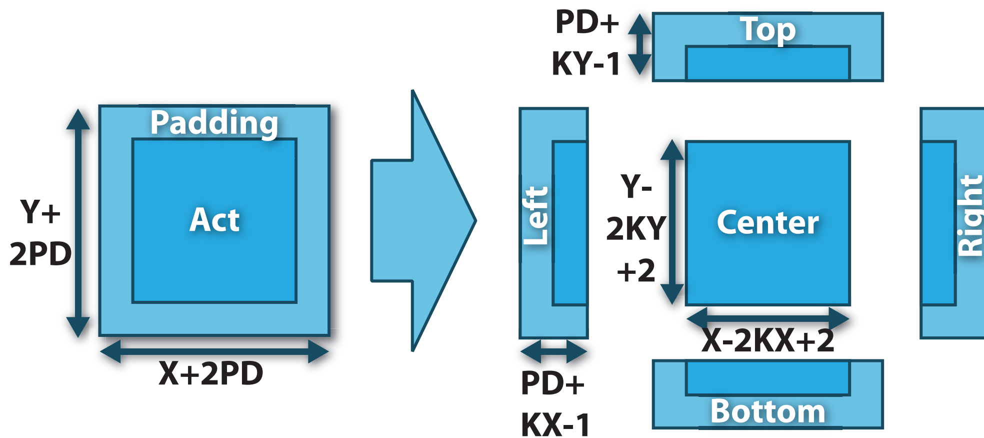


Figure 2.10: Convolution splitting into padded and non-padded regions for efficient computation.

are implemented as follows:

- Max pooling layers - to enable on-the-fly pooling, we process the outputs of convolutional layers in groups matching the pooling patch size. For each patch, we keep a running maximum updated as outputs are calculated.
- Batch normalization and binarization - Umuroglu et al.[47] observed that in binarized networks, batch normalization followed by binarization can be reduced to a thresholding operation followed by a conditional sign change. By using this approach only one floating-point parameter needs to be stored per kernel. Signs can be stored in binary-packed format and applied in batches using bitwise XNOR operations.

The idea behind fused kernels and threshold Batch Normalization is shown in Figure 2.11.

2.3.4 ARM NEON Support

For devices with ARM Cortex-A processors, like Raspberry Pi, we utilize NEON SIMD extension [148] to leverage hardware popcount support. Since our NEON-optimized functions

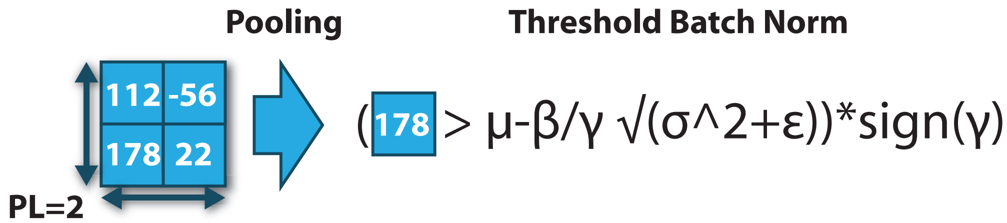


Figure 2.11: A 2x2 fused Max Pooling followed by threshold Batch Normalization.

use double- and quad-word (64- and 128-bit) support, we restrict sparsity to have a number of 32-bit packs that is a multiple of 2 or 4 for every kernel. Special care needs to be taken with padding in sparse convolutional layers, where within a single 64-/128-bit block, some packs may be within the padded region, while others are not. There is no easy way of skipping computation in that case, so we mask multiplication results that fall under the padding region. We use masking operations before popcounts to correct for that.

2.3.5 Binarization of the First Layer

In the case of image processing, usually, the only layer with a depth that is not a multiple of 32 is the first one, which makes it problematic to implement using depth-first convolutions. However, binarized implementations generally compute the first layer using full precision activations and binary weights to retain accuracy [141]. MLPs are used for MNIST, which is almost black and white, so we binarize the inputs and keep the first layers as XNOR layers. For CNNs, we refrain from sparsifying the first layer due to limited storage benefits.

2.4 Experimental Setup

Hardware and software platforms, benchmark datasets, and neural network architectures we use for our experiments are outlined below.

2.4.1 Platforms

We test our implementation on three different embedded development platforms from the STM Nucleo family [149] and a Raspberry Pi, with the configurations shown in Table 2.1. Our implementation is written in C and compiled with ARMCC version 5.06 for Nucleo devices, and GCC version 5.4 for Raspberry Pi, with -03 compiler optimizations enabled. For Cortex-M7 and M3 devices ("NUC Large" and "NUC Medium"), we use the built-in cycle counter to measure execution time over ten runs for each network [150]. For Cortex-M0 ("NUC Small"), we use the SysTick counter with a period of 1ms over 100 runs [151]. For Raspberry Pi, we use C library routines to measure execution time over 100 runs. We use an off-the-shelf USB power meter with two decimal digit precision to perform rough power measurements for each board.

2.4.2 Benchmarks

We evaluate our approach using two fully-connected networks (MLP) and a small convolutional neural network (CNN) on MNIST dataset, and two convolutional neural networks on CIFAR-10 and SVHN dataset as shown in Table 2.3. All datasets are image classification datasets with ten classes. MLP-Large (MLP-L) and CNN-Large (CNN-L) are used in [141]. CNN-Medium (CNN-M) uses the same convolutional layers as CNN-L but only has one fully-connected layer. CNN-Small (CNN-S) is a modified version of LeNet in [152], and MLP-Small (MLP-S) is a minimally sized MLP with one hidden layer. We also tested the same CNN-M for Google Speech Command dataset [145]. Networks are trained with PyTorch 0.4.1 [142]. All models are trained on the training set provided, and accuracies are measured on the testing sets. We used Adam optimizer [153] for training, with batch size of 256 and initial learning rate of 0.001.

2.4.3 Baseline

On Nucleo boards, we use the ARM CMSIS-NN [95], version 5.3.0, optimized for Cortex-M processors. CMSIS-NN uses a DSP extension present M7 processor for SIMD multiplication and accumulation. On Raspberry Pi, we use Arm Compute Library [154], version 18.03, with NEON extension enabled, -O3 compiler optimizations, no batching, and no multithreading for a fair comparison with 3PXNet. For both CMSIS-NN and Compute Library, we use 8-bit precision: q7_t and QS8 datatypes, respectively. The first layer in CNNs is implemented using CMSIS-NN/CL routines with binarization overhead since they are not packed and sparsified. We tried comparing two existing dense binarized implementations: BMXNet [59] and EBNN [63]. Unfortunately, we were not able to extract and compile underlying C implementations for BMXNet, and the eBNN runtimes we obtained were worse than CMSIS-NN 8-bit precision[1], therefore we refrain from reporting them. All results are generated for a batch size of $B = 1$. Larger batch sizes increase storage requirements and can make the models even more prohibitive to deploy on resource-constrained devices.

2.5 Results and Discussion

We discuss results for 3PXNet accuracy separately from performance, as only the latter depends on the hardware platform.

2.5.1 Accuracy & Model Size

Figure 2.12 compares accuracy vs. model size of 3PXNet to a dense binarized network eBNN [63]. 3PXNet achieves up to 4X (2.5X) reduction in MLP (CNN) model size with the same or better accuracy than eBNN. While both implementations use quantization, for eBNN,

[1] eBNN was $\sim 8X$ and $> 80X$ slower than our dense implementation for FC and CNN layers respectively, which we believe is partially because it uses 8-bit packs for higher flexibility, but lower computational efficiency, and using floating-point accumulation.

additional size compression comes from tweaking the network structure itself, whereas for us, it comes from pruning. In this work, we only explore the interaction binarization and pruning, but we plan to extend our approach to also optimize across the network structure itself.

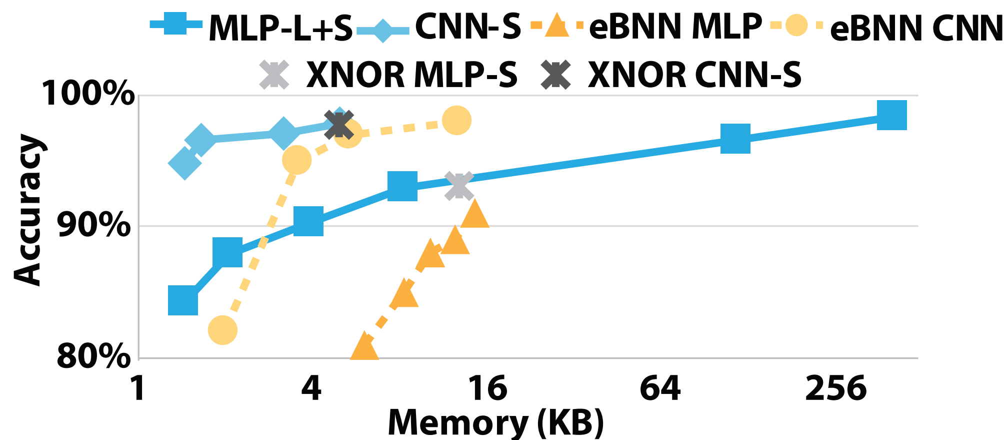


Figure 2.12: Accuracy vs. Memory tradeoff compared to eBNN and dense XNOR.

Training results are shown in Table 2.4. Each network is trained with two sparsity levels, and the size of the network is shown in parenthesis next to accuracy. For most models, the 3PXNet$_{low}$ has target sparsity of 90%, while 3PXNet$_{high}$ is targeting 95%. For MLP-L on MNIST, 3PXNet$_{low}$ has target sparsity of 95%, while 3PXNet$_{high}$ is targeting 99%. Output layers of CNN-M and CNN-L are kept at 50% sparsity due to their negligible impact on storage and computation. Because we are not pruning entire neurons, batch normalization layers are also not pruned. Since these constant-sized layers take up a varied proportion of the entire model and do not decrease in size with higher sparsity, the actual compression rate varies a lot across models, and we think it is more clear to list the actual sizes of the models, which are shown in parenthesis next to accuracy numbers. For binary/3PXNet implementations, weights are binary or ternary depending on the sparsity of a layer, and activations are all binary except for inputs to the first layer in CNNs. While for some of the networks, there is a noticeable drop in accuracy when comparing 8-bit with 3PXNet, as in the case of MLP-S on MNIST and networks on CIFAR-10, we argue that the main benefit of

3PXnet is enabling deploying some of those models on heavily memory constrained devices, which would not be possible with 8-bit and, in some cases, even dense binary. As shown in Table 2.5 and 2.6, binarization enables implementation of the network in 3 cases, and 3PXNet enables another 2. When the model loses noticeable accuracy when binarized, it tends to lose more when pruned. Accuracy loss can be mitigated by using more permutations per layer in addition to the "free" one or using smaller pack sizes like 8. For MLP-S on MNIST, reducing pack size to 16 and 8 for "3PXNet$_{low}$" increases accuracy to 88.42% and 90.09%, respectively. The added indexing storage and runtime overhead are usually not a good trade-off when naively using packs of size smaller than 32, but we plan on exploring more efficient implementations of such networks in the future.

Table 2.4: Accuracy and network size (KB, in brackets) comparison.

Dataset	MNIST			CIFAR-10		SVHN		Speech
Model	MLP-L	MLP-S	CNN-S	CNN-L	CNN-M	CNN-L	CNN-M	CNN-M
8-bit	98.67% (36.9k)	98.28% (102)	99.42% (32.7)	92.52% (14.1k)	92.51% (4.69k)	95.65% (14.1k)	95.15% (4.69k)	97.87% (4.70k)
XNOR	98.40% (4.64k)	93.15% (13.1)	97.83% (4.46)	89.07% (1.80k)	88.29% (592)	95.03% (1.80k)	95.00% (592)	96.64% (591)
3PXNet$_{low}$	98.37% (421)	90.35% (3.96)	96.60% (1.66)	84.74% (257)	82.49% (98.8)	93.51% (257)	92.57% (98.8)	94.32% (97.6)
3PXNet$_{high}$	96.58% (120)	87.93% (2.08)	96.27% (1.46)	81.40% (143)	78.28% (61.7)	92.25% (143)	90.61% (61.7)	92.81% (60.5)

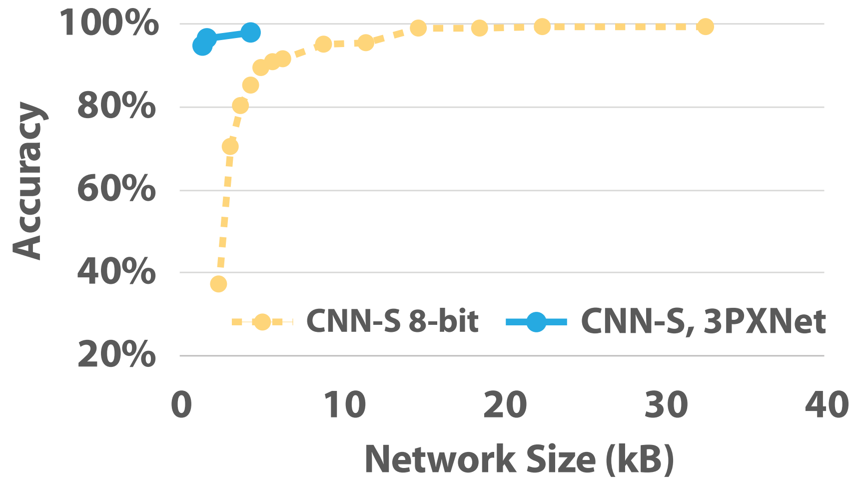
Network pruning can also be performed on higher precision models, so we compared our performance to magnitude-based weight pruning [82]. Apart from the processing difficulties resulting from irregular sparsity, the index of each active weight also needs to be stored. Since the size of a filter can easily surpass 255, which is the limit of 8-bit indexing, we used 16 bits for each index, but Figure 2.13 shows the accuracy comparison between sparse 8-bit networks and binarized networks. For MNIST, sparse 8-bit networks have a worse accuracy-size trade-off compared to dense binarized networks, let alone the sparse ones. For CIFAR-10, sparse 8-bit networks have higher accuracy for the same size compared to dense binarized networks but are worse than 3PXNet implementations. To achieve the model size of binarized networks, an 8-bit fixed-point network requires very high sparsity, particularly

due to the indexing required and loses too many connections to sustain accuracy. Methods to reduce indexing overhead for fixed-point networks are proposed in [21] but are mostly limited to the packing of 2, so the conclusion doesn't change. Under the size constraints of very small microcontrollers, 3PXNet offers better accuracy-size trade-offs, even when not accounting for the benefits in processing efficiency our structured pruning method brings.

2.5.2 Performance & Energy

Runtime and energy results for 8-bit, XNOR, and 3PXNet are shown in Tables 2.5 and 2.6 for MNIST and CIFAR-10/SVHN/Google Speech Command, respectively. We report only one result for CIFAR-10, SVHN, and Speech Command. The first two have the same network structure, which yields the same runtime. For Google Speech Command, only the first and the last layers are different, and runtime differences are so small (¡5%) we opt not to report them separately. For the MLP-S 3PXNet shows between 10.7x and 25.2x runtime improvement over CMSIS-NN, and 1.8x to 3x over Dense XNOR implementation. MLP-L can fit the NUC Large only after binarizing and sparsifying the network. On Raspberry Pi, the speedups obtained for both MLP-S and MLP-L are much higher than expected (76x-400x). For the former, the model is so compact that overheads like memory management limit the performance of 8-bit (ARM CL) implementation. For MLP-L, which has a model size in tens of MBs, the large latency might be caused by memory bandwidth limitations. This result means that dense and sparse binary implementations can provide speedups beyond 8x (vs. 8-bit binary) on memory bandwidth-constrained models and architectures.

On CNN-S, 3PXNets show between 2x and 2.6x improvement over 8-bit implementations, and between 1.06X and 2.7x improvement over dense XNOR, on Nucleo platforms, for low and high sparsity, respectively. On Raspberry Pi, the speedups are 2.6x and 2.8x versus 8-bit, and 1.5x and 1.6x versus dense binary. 3PXNet CNN-M is 2.2x and 2.7x faster than dense binary, on the largest Nucleo device, where the 8-bit model cannot fit altogether. On Raspberry Pi, 3PXNet achieves a 7.4-10.4x speedup vs. ARM CL and a 2-2.75x speedup



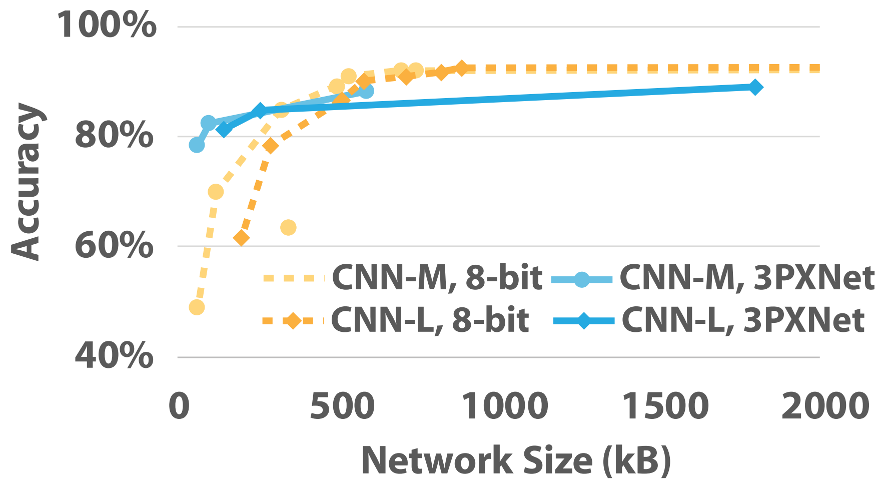


Figure 2.13: Accuracy comparison between sparse 8-bit network and 3PXNet, for MNIST (a) and CIFAR-10 (b).

over dense binary. Energy reduction is proportional to runtime, which means 3PXNets could greatly extend the battery life of edge devices.

A few factors limit achievable speedups, for both dense networks and 3PXNets. First is the lack of hardware popcount support on Cortex-M processors, which is the case for most embedded microcontrollers. Even heavily optimized software implementation, which we use, is quite inefficient [155]. On the small MLP, especially sparse versions, pure XNOR speedups are further limited by overhead of input binarization and batch normalization, particularly on NUC Medium and Small platforms, which don't have hardware floating point units (FPUs). We plan on exploring fixed-point batch normalization in the future to alleviate that. CNN speedups are heavily limited by the binary-weight first layer, especially for CNN-S, e.g., dense XNOR on "NUC Large" spends 54% of total runtime in the first layer, whereas for $3PXNet_{\text{high}}$ that number reaches 92%. In principle, binary-weight layers have an advantage over full-precision ones because multiplication can be removed with an up/down counter. However, this comes at a cost of introducing data-dependent conditional statements, which are potentially worse than actual multiplication. An alternative is to preserve multiplication and compressed weights, which achieves storage reduction at a very small cost to runtime due to weight unpacking overhead. We plan on implementing more efficient BWN layers in future work. Finally, 3PXNet performance improvements over dense XNOR implementations are limited by indexing-related overheads.

Table 2.5: Runtime (ms) and energy (mJ, in brackets) for MNIST networks. A dash indicates a given model could not fit in memory.

	MLP-S				MLP-L				CNN-S			
	8bit	XNOR	$3PXN_{low}$	$3PXNet_{high}$	8bit	XNOR	$3PXNet_{low}$	$3PXNet_{high}$	8bit	XNOR	$3PXNet_{low}$	$3PXNet_{high}$
RPi	2 (9.5)	≈5e-3 (0.02)	≈5e-3 (0.02)	≈5e-3 (0.02)	191.3 (908)	4.9 (23.2)	2.5 (11.8)	0.41 (1.9)	12.4 (58.9)	7 (33)	4.8 (23)	4.5 (21)
NUC Large	1.83 (2.03)	0.37 (0.41)	0.17 (0.19)	0.14 (0.15)	—	—	10.1 (11.2)	3.97 (4.41)	47.4 (52.6)	34.09 (37.8)	23.4 (26)	23.2 (25.7)
NUC Medium	19.9 (8)	2.4 (0.9)	1.05 (0.4)	0.79 (0.3)	—	—	—	—	1733 (693)	731.2 (292)	676.8 (271)	673.3 (269)
NUC Small	—	2.9 (0.7)	1.6 (0.4)	1.3 (0.3)	—	—	—	—	—	1176 (294)	1109 (275)	1102 (277)

Table 2.6: Runtime (ms) and energy (mJ, in brackets) for CIFAR-10/SVHN/Speech networks. A dash indicates a given model could not fit in memory.

	CNN-M				CNN-L			
	8bit	XNOR	$3PXNet_{low}$	$3PXN_{high}$	8bit	XNOR	$3PXNet_{low}$	$3PXNet_{high}$
RPi	551 (2.6k)	146 (700)	74 (350)	53 (250)	638 (3k)	154 (730)	75 (350)	54 (250)
NUC Large	—	3625 (4k)	1630 (1.8k)	1346 (1.5k)	—	—	1632 (1.8k)	1347 (1.5k)

2.6 Distinction between 3PXNet and Ternary Networks

Traditional implementations of ternary networks, which restrict values to -1, 0, and +1, can be considered binary-sparse, like 3PXNet. However, they store numbers in 2-bit format and don't skip computation, which makes it impossible to capitalize on the benefits of either binarization (XNOR multiplication), or sparsity (storage and computation reduction) [40, 156, 157, 158, 159, 160, 161]. Thus we would like to differentiate between explicitly Ternary Networks, using 2-bit representation, and Binary-Sparse Networks, leveraging XNOR multiplication and size compression, like ours. As an example of the latter, Lin et al. [162], exploited Singular Value Decomposition to reduce kernel sizes in BNNs. Certain software and hardware implementations rely on operand-gating XNOR multiplication [163, 164], however they still require 2-bits of information per weight: value and mask. Li and Ren [165] decomposed first-layer activations into "bit-slices" and explore pruning opportunities in those, but their scheme does not extend over the whole network. Faraone et al. [23] have discussed implications of exploiting sparsity in binarized FPGA accelerators, but not software ones.

2.7 Conclusion

We have developed the first software implementation and corresponding training methodologies for sparse binarized networks or 3PXNets. 3PXNets can deliver up to 300x (38x) smaller model sizes compared to 8-bit fixed point (dense binarized) networks allowing us to fit com-

plex deep learning models on smallest microcontrollers for the first time. Even in smaller models that can fit with conventional approaches, 3PXNets achieve up to 25x improvement in runtime and energy. We show multiple sub-ms and sub-mJ models on commodity low-end microcontrollers, which would not be possible without 3PXNet. We release 3PXNet as an open-source library targeting machine learning at the edge.

CHAPTER 3

ACOUSTIC - Accelerator Built on Randomness

High demand for edge machine learning inference is leaving resource-constrained devices struggling to cope with large and computationally complex models. Software implementations, like the one described in Chapter 2, while useful, cannot overcome the fundamental limitations of the underlying hardware. Even the increasingly frequent use of domain-specific acceleration cannot offset their limitations. However, there is enormous potential in exploiting data reuse opportunities and amenability to reduced precision as a means to improve performance on such devices. This potential has been left mostly untapped, as it requires a level of compute density unattainable for conventional fixed-point arithmetic. Stochastic Computing, an approximate computing paradigm, would be capable of delivering such densities if it was not for multiple underlying precision issues. In this work, we present ACOUSTIC: a framework for training and architecture design, offering scalable, fully-stochastic, high-density CNN inference. By taking full advantage of SC compute density, ACOUSTIC architecture delivers server-class parallelism within a mobile area and power budget - a $12mm^2$ accelerator can be as much as 38.7x more energy-efficient and 72.5x faster than conventional fixed-point accelerators. It can also have up to 79.6x lower energy consumption than state-of-the-art stochastic accelerators. At the lower end, ACOUSTIC achieves 8x-120x inference throughput improvement with similar energy and area when compared to recent mixed-signal and neuromorphic accelerators. ACOUSTIC can also take advantage of runtime-configurable precision at minimal hardware cost, enabling tradeoffs between latency and accuracy - something which is impossible for conventional fixed-point architectures. To prove the feasibility of our approach, we show both FPGA and ASIC implementations

of a lower-end ACOUSTIC configuration, with performance competitive with much more resource-heavy architectures.

Collaborators:

- Tianmu Li, Electrical and Computer Engineering, UCLA.
- Rahul Garg, then Electrical and Computer Engineering, UCLA.
- Tristan Melton, Electrical and Computer Engineering, UCLA.
- Professor Sudhakar Pamarti, Electrical and Computer Engineering, UCLA.
- Professor Puneet Gupta, Electrical and Computer Engineering, UCLA.

3.1 Introduction

Rapidly growing demand for deep learning algorithms, coupled with their immense computational and memory requirements, has made them an enticing target for custom-built accelerators [29, 2, 15, 166, 90, 14, 167, 26]. Energy and area optimization of these accelerators is especially crucial for resource-constrained edge devices, which are taking on an increasing share of inference workloads due to privacy, energy, and latency concerns. One of the notable contributions towards making deep learning accelerators more efficient was the realization that reducing the precision of underlying computation incurs very little to no accuracy degradation [168, 141, 37, 19, 169] when we train these networks appropriately. Another was recognizing vast data reuse opportunities present, particularly in convolutional neural networks (CNNs), and devising dataflows that take the best advantage of those [13, 130]. Both low precision and data reuse are crucial for reducing the number of memory accesses, on- and off-chip, which are the critical contributors to overall system energy consumption [12].

The above realization has paved the way for the renaissance of stochastic computing, or

SC for short. It is a number representation system half a century old that has long been abandoned by mainstream research due to its inherent approximation issues [100]. While it may never be suitable for applications that require floating-point precision, it is exceptionally competitive for sub-8-bit fixed-point numbers [170]. Coupled with the unparalleled levels of compute density it provides, SC can be an excellent candidate for convolutional neural network (CNN) acceleration, a realization that spawned multiple works in recent years [171, 106, 172, 166, 105]. However, researchers have not been able to harness the full benefits of SC. Specific issues, like accuracy degradation when performing addition, force early conversion or even abandoning stochastic accumulation altogether, reducing SC domain to just multiplication [171, 106, 172]. Further, most prior SC works develop network-specific ASICs (e.g., [106]) rather than programmable accelerators.

This Chapter presents ACOUSTIC - Accelerating Convolutional neural networks through Or-Unipolar Skipped sTochastIc Computing, a scalable, programmable, and fully-stochastic CNN accelerator framework. ACOUSTIC integrates multiple algorithm and architecture optimizations that allow us to harness the full benefits of SC:

- **Optimization of SC primitives for deep neural networks.** We develop a spatially or temporally processed split-unipolar stochastic representation that enables 2X+ shorter streams while retaining the bipolar capability of representing both positive and negative weight values required for neural network inference. Second, we enable practical stochastic accumulation through the use of OR-based accumulation, which has unique scale-free saturating addition properties with unipolar SC streams critical for deep neural networks with extensive accumulations. OR-based accumulation can reduce MAC size by 4X-20X or more compared to fixed-point accumulators used by most existing SC approaches while retaining comparable accuracy.

- **Computation skipping method for stochastic average pooling** that can deliver latency and energy reduction proportional to the size of the pooling window (4X-9X)

on the convolutional layer itself. Computation skipping is made possible through fully-stochastic accumulation.

- A **scalable, configurable, fully-stochastic convolutional neural network accelerator** architecture built to harness SC compute density to maximize activation and weight reuse while minimizing or completely removing partial sums to reduce conversion overheads and the number of memory accesses required. The compact size of our MAC enables levels of compute density unachievable for other SC accelerators, not to mention fixed-point binary ones.

- **Training optimization**, which allows it to adapt to the characteristics of SC computation. Through modifications to the accumulation kernel and noise injection, stochastic OR-based addition can be used without significant accuracy loss, something that was deemed impossible in previous works. Furthermore, this allows us to improve the scalability of training of networks for SC accelerators improving the training runtime by orders of magnitude.

- **Hardware support for runtime-configurable accuracy and latency trade-offs**. Through minimal hardware overheads, our architecture can support configurable stream lengths on a layer-by-layer basis.

- **FPGA and ASIC performance evaluations**. We show actual results on a synthesized RTL code running on the AVNET ULTRA96-V2 platform, as well as a taped-out 14nm custom chip, proving the feasibility of our design.

3.2 ACOUSTIC Optimizations for DNNs

3.2.1 Split-Unipolar Representation

To capitalize on both shorter streams offered by unipolar representation, as well as the ability to represent negative weights, we propose the split-unipolar representation. It uses two streams to represent each weight, one for the positive and one for the negative component. For a positive weight value, its corresponding negative stream is 0, and vice-versa. Because activations (inputs) of a neural network layer are typically non-negative due to the ReLU activation function [79, 30, 1] in the previous layer, they can be represented using only a single positive stream. The activation streams are multiplied and accumulated separately with positive and negative weight components using up counters, whose values are then subtracted from each other to obtain the final result. Since the counter output is in the fixed-point binary domain, ReLU activation is easily implemented as a bitwise AND of the inverted sign with every other bit. There are two ways in which this method can be realized in hardware - using spatial or temporal unrolling, as shown in Figure 3.1a).

Spatial unrolling doubles the compute arrays, and shares both weights and activations between them. Weights to one of the arrays are masked using their sign, and weights to the other are masked using the inverse of the sign. Each array has its output counters, and corresponding results from both arrays need to be subtracted. While this results in overall 50% utilization, increasing it for any arbitrary network is non-trivial, since the positive and negative weight distribution is not known a priori. A simple example of spatially unrolled split-unipolar computation is shown in Figure 3.1c). A similar idea has been recently, independently proposed by [110], however, it requires knowing the positive/negative distribution of weights a priori.

Temporal unrolling achieves the same result with a single MAC array, where computation happens in two phases. In the first phase, all negative weights, and by extension their respective multipliers, are gated using their sign. This means only results corresponding to

positive weights are accumulated, and the output counters count up. In the second phase, the mask is inverted, only negative weights contribute to the outputs, and counters count down. Supporting temporal split-unipolar representation requires only a couple of additional gates per SNG and replacing regular up-counters (CNT) with up/down counters (U/D CNT). A simple example of a 2-wide MAC with one positive and one negative weight, and stream length of 8 is shown in Figure 3.1b).

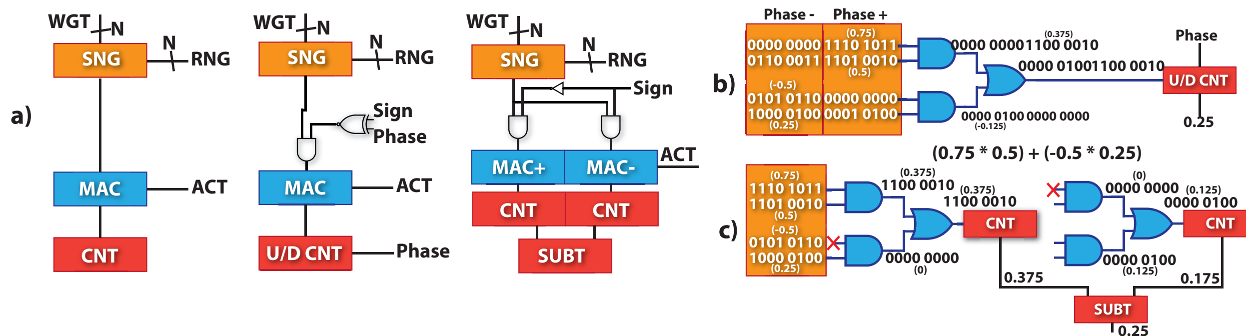


Figure 3.1: Circuit level support for unipolar, temporal split-unipolar, and spatial split-unipolar representations (a) and an example of 2-wide split-unipolar MAC temporarily- (b) and spatially-unrolled c).

3.2.2 OR-based Scaling-Free Accumulation

As mentioned in Section 1.2.2, OR accumulation has been largely disregarded by prior work due to its inherent inaccuracy for bipolar streams. Since the use of split-unipolar representation eliminates the need for bipolar streams, we decided to revisit OR-based addition. Figure 3.2a) shows the accuracy comparison between MUX and OR for accumulating $3 \times 3 \times 256 = 2304$ random numbers. Since neural networks generally require large matrix multiplications that have extensive additions, MUX is not suitable for compact SC addition due to its high absolute error at even moderately long bitstream lengths. As mentioned earlier, using fixed-point binary adders is undesirable due to a large area overhead.

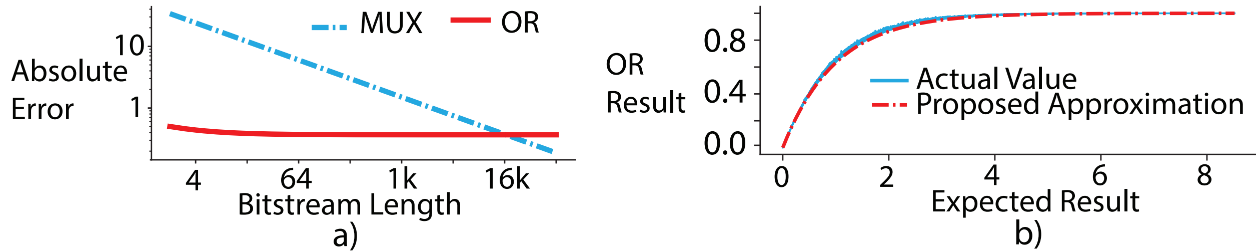


Figure 3.2: Accuracy comparison between MUX and OR a) and comparison of approximation methods for OR accumulation b).

Though OR accumulation has a systematic error, it is well-defined and, therefore, can be taken into account by replacing all additions with OR-addition during the training of a neural network. It requires multiplications in the neural network to be performed explicitly while training, and also slows down addition during both forward and backward pass (∼15X longer training runtime). To speed up training, we approximate the effect of OR addition using the following function:

$$OR(a_1, a_2, ..., a_n) \approx 1 - e^{-s} \tag{3.1}$$

Where s is the sum of inputs. This equation approximates OR-addition by adding an activation function after the normal network layer. To show the validity of this approach, we extracted runtime results from training with OR-addition, with shown in Figure 3.2b). OR-additions have minimal variations during run time, and the proposed method provides a good approximation across the entire range. Using OR accumulation for the output layer slows down training due to the limited range. To offset this effect, we add a batch normalization layer after the output layer to be computed by the host. Since it is done only after the final layer, it does not require offloading intermediate results, which would have a significant impact on runtime.

3.2.3 Computation Skipping using Stochastic Average Pooling

While multiplexer-based average pooling has been used in prior SC work [173, 105], to the best of our knowledge, we are the first to exploit its unique properties to perform computation skipping. The average pooling multiplexer selects inputs based on a random select signal, same as scaled addition. To get the required output value, the select signal does not need to be random as long as the inputs are random and independent from each other. Since we know the bits the multiplexer "chooses" a priori, we can skip computation for all other inputs' bits. Instead of passing multiple streams through the pooling multiplexer, we concatenate shorter streams, either in the stochastic or fixed-point binary domain. This allows us to reduce the computation required by the convolutional layer preceding a pooling operator by 4x to 9x, depending on the pooling window size. The area overhead of supporting computation skipping is minimal - it increases the size of an individual activation counter by 2.7% to 8.7%, depending on the pooling window size, which is $< 1\%$ of the overall accelerator area. The issue with the computation skipping scheme is that the results are correlated, thus necessitating the randomization of outputs. However, ACOUSTIC architecture converts the streams to binary after each layer, removing the correlation problem.

Of course, the accuracy reduction of using average pooling needs to be addressed. Table 3.1 shows the accuracy comparison between average and more commonly used max pooling methods for two models and datasets. We used the pre-trained models from PyTorch [142] for max pooling, then replaced pooling layers with average pooling and retrained the models. We believe this small accuracy increase does not justify the area and power penalty incurred by using max pooling, especially when factoring in the benefits offered by computation skipping.

3.3 ACOUSTIC Architecture

In this section, we motivate and develop the ACOUSTIC accelerator architecture.

Table 3.1: Accuracy comparison between different pooling methods.

Network	Dataset	Max Pooling	Average Pooling
Small CNN [95]	CIFAR-10 [174]	80.61%	80.53%
AlexNet [27]	ImageNet [28]	79.09%	78.87%

3.3.1 Understanding SC Benefits

While the prospect of implementing a multiplier using a single gate might seem area, and energy-efficient, it is far from the complete picture. Stochastic streams can require orders of magnitude more bits than equivalent fixed-point representations, which has a profound impact on energy and latency of the computation. Further, that length makes it prohibitively expensive to store values in their stream representation. This forces conversion to and from fixed-point domain when storing and reading values from memory, respectively. All of the above needs to be considered when evaluating area, energy, and latency cost of SC-based computation. At the same time, the impact of conversion can be reduced through data reuse, which also needs to be considered. To calculate the actual per-MAC cost, we have synthesized individual blocks: 8-bit fixed-point binary and stochastic MACs, 8-bit registers, and LFSR-based SNG using a commerical 28nm library and Synopsys Design Compiler synthesis tool. We assumed that RNGs are shared among multiple SNGs, which is a common approach to amortize their cost [175, 108].

For Stochastic Computing based MACs, we consider two types of reuse. First is the output reuse, which amortizes stochastic-to-fixed-point (SC2FXP) conversion. It can be facilitated by performing extensive reduction operations in parallel in the stochastic domain. The cost of SC2FXP conversion does not scale with the width of accumulation as long as all of the computation is performed in parallel, since only a single output stream needs to be converted. SC2FXP spatial output conversion is different from the temporal output reuse employed by classical accelerators, where a partial sum is kept locally, while serially

accumulating the final result over multiple clock cycles [13]. The second type of reuse we consider is input, or fixed-point-to-stochastic (FXP2SC), reuse, where a single input can be read from memory and converted once, and reused across multiple MAC operations. In the context of convolutional neural networks, this can be applied both to activations and weights. FXP2SC reuse is equivalent to input reuse in conventional accelerators; however, only spatial reuse is possible [13]. Temporal, or sequential, reuse for SC accelerators would require buffering the streams itself, which is prohibitively expensive.

For this experiment, we considered three different input reuse patterns: no input reuse, one of the inputs (e.g., activations) reused across 32 MAC operations, and both inputs (e.g., activations and weights) reused across 32 and 16 MAC operations, respectively. Figure 3.3 shows that even with input and output reuse when all of the costs are accounted for, SC MACs can, at best, achieve parity with fixed-point in terms of energy. While our proposed optimizations, such as split-unipolar representation and computation skipping, can significantly improve SC's energy efficiency, it is not where it excels.

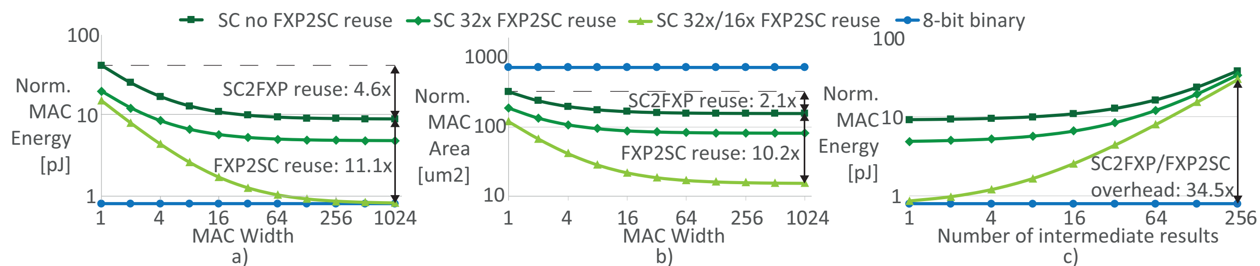


Figure 3.3: Normalized MAC energy (a) and area (b) for 8-bit fixed-point and 256-long, unipolar SC implementations in TSMC 28nm node with 200MHz clock, with different data reuse patterns. Normalized MAC energy for 256-wide MAC when intermediate results are converted to binary (c).

Figure 3.3b) shows where the true benefit of SC lies - compute density. Even without any data reuse, stochastic MACs are more than two times smaller than their fixed-point

equivalents. With reuse, SC implementations can be up to 47x more compact. While high compute density directly improves system cost through reduced area footprint, it is much more valuable as a means of reducing overall energy consumption and inference latency. As multiple previous works have shown, the bulk of energy in contemporary computing systems goes into DRAM and on-chip memory accesses, not the actual computation [12, 2, 130]. High compute density made possible by SC enables higher levels of parallelism, which, in turn, allows us to amortize the costs associated with memory accesses and conversion. To do that, however, ample data reuse patterns need to be present, which is why convolutional neural networks are an ideal application for SC-based acceleration.

This conclusion leads us to two essential dataflow guidelines for SC-based accelerator design: (1) avoid partial sums and (2) maximize compute density. To understand the first point, consider the processing elements in conventional fixed-point binary accelerators. They often rely on tightly coupled local scratchpads for efficient, partial sum buffering for output reuse [13, 2, 130]. The same approach in an SC-based accelerator would require either buffering the intermediate result in its stochastic stream form or converting it back to a fixed-point binary representation. The former can drastically reduce the compute density: buffering an intermediate 128-bit stream for a 32-wide MAC block would increase its area by 10.6x. The latter, while possible, can very quickly cause a substantial increase in MAC energy consumption as the number of intermediate results grows, as shown in Figure 3.3 c) for a 256-wide MAC. It is, therefore, imperative to minimize or eliminate partial sums and amortize conversion costs through more parallelism. SC-DCNN/HEIF [106, 105] achieve this through completely unrolled, parallel computation. Their approach, which requires a custom design for every network topology, results in a prohibitively large area, making it highly impractical. Other approaches, like SCOPE or BISC-MVM, rely on the accumulation in the fixed-point binary domain, limiting achievable compute density [107, 166].

Our second guideline, maximizing compute density, has important repercussions for the on-chip connectivity. Many conventional fixed-point architectures put an emphasis on flexi-

ble network-on-chip (NoC) design to better facilitate data reuse [13, 2, 130]. For SC, however, using sophisticated NoCs can drastically reduce achievable compute density. Consider the following scenario: to add a minuscule level of configurability, we allow each MAC block to have two possible sources of activation streams. It would require using a 2:1 multiplexer, which at least doubles the area and energy of the MAC block. It is easy to see how an elaborate NoC could quickly dominate the area of an SC-based accelerator, reducing the precious compute density. While there exist successful accelerators relying on rigid connectivity schemes like systolic arrays, those are not easily amenable to SC computation. Reconciling the streaming nature of both SC and systolic arrays would require either extensive buffering of stochastic streams, or very complex multi-cycle path design.

3.3.2 Accelerator Architecture

3.3.2.1 Overview

Figure 3.4a) depicts the overall block diagram of the ACOUSTIC Accelerator. It includes two external interfaces: configuration ① and direct memory access (DMA) ②. The former is used to load the machine code representing a given network model to the instruction memory and enable the processing. The execution control scheme is described in detail in Section 3.3.2. The DMA interface loads the initial activations and weights for each neural network layer from external memory to their respective on-chip memories, ③ and ④ and writes out final classification results. ACOUSTIC does not use a unified global buffer; instead activations and weights have their dedicated on-chip memories. While this organization might result in sub-optimal memory utilization, it allows us to better tailor each memory to its purpose in terms of width, depth, and bandwidth.

The following is a high-level explanation of the computation dataflow, which we will expand on in the following sections. There are three activation memories ③, corresponding to three MAC array columns. This organization provides optimal support for the commonly

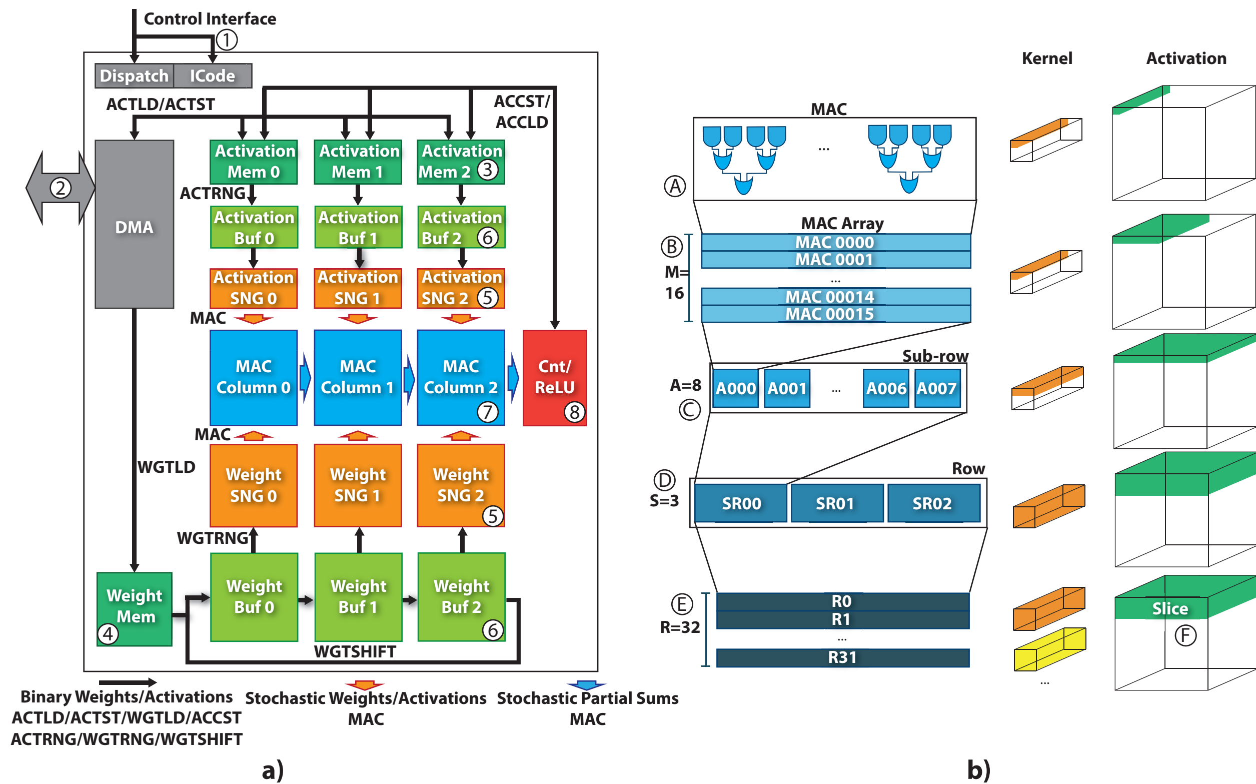


Figure 3.4: Block diagram of the proposed ACOUSTIC accelerator (a), and the hierarchical organization of the compute engine with parts of the kernel and activation tensors covered by each level of hierarchy (b).

used 3x3 convolutional kernels, as explained below. Each of the MAC array columns has a corresponding set of activation and weight SNGs ⑤, each with its dedicated value buffer ⑥. Those buffers are responsible for providing reference values to SNGs during the conversion process and do not have an obvious proxy in conventional accelerators. To avoid confusion, we only use the term "buffer" to refer to SNG buffers and not to any of the on-chip memories. It is worth noting that due to the combinational nature of SC arithmetic units, conversion, or SC stream generation, is equivalent to computation - when one is happening, so must the other.

Stochastic streams from the SNGs are fed into their respective MAC array columns ⑦, which perform mostly hard-wired multiplication and accumulation. Results generated this way are then passed through a flexible SC accumulation fabric supporting different layers and kernel sizes. Final sums are sent to the activation counter modules ⑧ for SC2FXP conversion, followed by the ReLU activation. Fixed-point activation implementation is both cheaper and more precise than the SC equivalent. When the computation, or stream generation, is completed, results from the activation counters are written back to one of the activation memories as inputs for the next layer.

The general processing flow will, therefore, require the following steps. (1) Load activations from external memory to on-chip activation memories (first layer only). (2) Load weights from external memory to on-chip weight memories. (3) Load activations from on-chip memories to activation SNG buffers. (4) Load weights from on-chip memories to weight SNG buffers. (5) Generate and compute bitstreams. (6) Store outputs from activation counters to activation on-chip memories. (7) Repeat steps 3-6 for each of the layers in the neural network model. It is important to note that some of the above steps, like activation and weight loading, can be overlapped, and ACOUSTIC is designed to take the best advantage of those opportunities to improve performance.

The compute engine of the ACOUSTIC accelerator has a highly hierarchical organization, shown in Figure 3.4b), to balance parallelism with flexibility, as per our guidelines. First,

hard-wired, multiply-accumulate units Ⓐ perform 96-wide multiplication and accumulation. This reduction spans the depth of the activation tensors. Those MAC units are the basic building blocks of our compute engine, and thanks to their rigid connectivity, they can form an extremely dense computing fabric. They are followed by a flexible accumulation fabric which maps partial sums onto specific outputs for further reduction. A group of M=16 MAC units with partially-shared activations and fully-shared kernel weights forms a *MAC array* Ⓑ. This grouping is equivalent to spatially unrolling a single kernel sliding horizontally across a partial activation tensor. Arrays, in groups of A=8, form a *sub-row* Ⓒ. Each of the arrays in a sub-row has its respective activation SNGs, all serviced from the same activation memory, with a single set of weight SNGs shared among them. Their grouping corresponds to extending horizontally the range of the activation tensor that is covered by a single kernel. We refer to those partial activation sub-tensors as *slices* Ⓕ. Sub-rows in groups of S=3 form *rows* Ⓓ, which extend the partial activation tensor in the height dimension. There are R=32 rows Ⓔ, which means 32 kernels can be computed in parallel using the same activations.

3.3.2.2 Layer Size

To understand how ACOUSTIC organizes the dataflow, let us first consider a convolutional layer shown in Figure 3.5a) and b). They show the intra- and inter-array mapping of the same layer, respectively. This layer has an input activation tensor of height and width y=x=128, and depth z=32 ①. It is convolved with a single kernel tensor of height ky=1, width kx=3, and depth kz=32. An activation slice of height y'=1, width x'=128, and depth z'=32 ② is loaded into a column SNG buffer which can hold 4096 values ③. Because the buffers are one-dimensional, the slice is flattened using depth (z) as the innermost loop. Typically, each of the three columns' SNG buffer would get a single slice to extend across the activation tensor height (y). In this case, since the kernel has a height (ky) of 1, only one input slice is needed at a time, so only one of the columns and sub-rows is used. If a slice cannot fit in the available buffers, it is partitioned in depth (z) dimension and processed sequentially

in multiple passes. Within the column SNG buffer, the slice is partitioned into 8 *partial slices* of 512 values, each corresponding to a tensor of a height y”=1, width”=16 and depth z”=32 ④. Each partial slice is an input to one of the eight arrays within the sub-row, as shown in Figure 3.5b) ⑤. Within each array, the first MAC unit receives the first 96 values of the partial slice and its kernel weights ⑥. All of the MACs in a given array use the same weights, but activations are offset by 32 between them. This offset simulates a sliding convolutional with the stride of 1 in width (x) dimension. Mapping layers with depths z¡32 requires special consideration and is discussed later. Each of the MAC units computes one adjacent output for a single kernel. The last two MAC units in an array are connected to the first 64 inputs of the subsequent array to ensure spatial continuity of the sliding window ⑦. We call those *overlap connections*. As a result, a single array will compute a partial output tensor of height oy’=1, width ox’=16, and depth oz’=1 ⑧. Across all eight arrays, this will combine to form an output tensor of height oy=1, width ox=128, and depth oz=1 ⑨. To extend the kernel size to height ky=3, three successive activation slices are loaded into three columns, and their corresponding outputs are accumulated together.

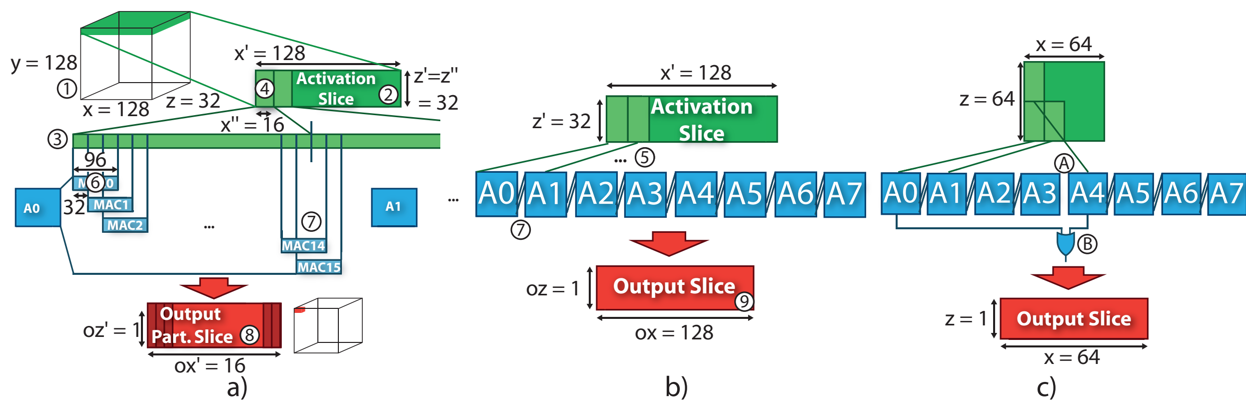


Figure 3.5: Convolving a 1x128x32 input slice with a 1x3x32 kernel to compute a 1x16x1 partial output slice a). Extension across multiple arrays to compute a 1x128x1 output slice b). Configuration for a 1x64x64 input tensor.

Throughout the CNN layers, the width and height of activation tensors decrease, through either pooling or striding, while their depth increases. ACOUSTIC's compute engine can easily support that by modifying the overlap connections, as shown in Figure 3.5c). When the depth increases, overlap connections between certain arrays are severed - for example, for a 1x64x64 activation slice, there is no connection between arrays 3 and 4, as the full extent of activation tensor width spans only four arrays Ⓐ. At the same time, outputs of the arrays corresponding to the same position are reduced across the depth (z) dimension - for a 1x64x64 slice, corresponding outputs of arrays 0 and 4 are accumulated together Ⓑ.

Input and output height is processed sequentially, as shown in Figure 3.6 a), for a single kernel. To compute the first output slice, three input slices are loaded into their respective columns ①. After the outputs are computed, the column corresponding to the uppermost slice is reloaded with the next slice, and kernel values are shifted between columns to ensure kernel and input slices are aligned properly ②, imitating a vertically sliding convolutional window. Then the second output row can be computed ③. The advantage of this approach is that only one of the columns' activation buffers needs to be reloaded between subsequent computational phases. All rows will share the same activations, but each one of them will have its own set of weights corresponding to a single kernel, as shown in Figure 3.6b). Outputs of different rows are transposed, with width and depth dimensions being swapped, which creates the depth-first layout required on the input side of the next layer. This transposition guarantees a self-aligning data layout, which is consistent between layers. The maximum number of outputs that can be computed at a time is 4096, which is the number of output counters.

3.3.2.3 Kernel Size

Supporting smaller kernel sizes is straightforward. In the simplest form, only one or two columns will be used to compute 1x1 or 2x2 kernels. More columns can also be used to compute multiple 1x1 kernels concurrently in the same row, as long as there are enough

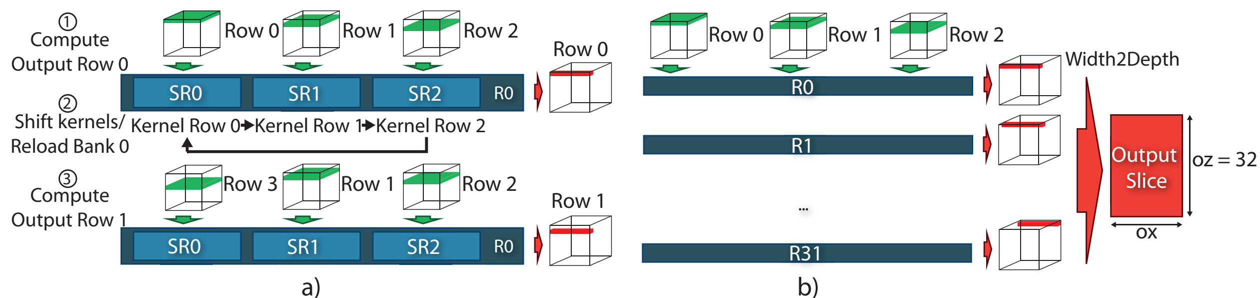


Figure 3.6: Processing two successive output rows sequentially (a), processing multiple kernels at the same time, with output transposition (b).

output counters for a given layer. To support larger kernel sizes, we couple outputs of multiple rows, as shown in Figure 3.7a), for kernels up to 6x6. First MAC unit in row 0 gets values corresponding to the first 1x3x32 partial slice, and the 4th MAC unit in row 1 gets the second 1x3x32, which means together they are covering a 1x6x32 partial input slice. By splitting the kernel across rows 0 and 1 and adding their corresponding results, 3x4, 3x5, and 3x6 kernels can be computed at the same time. To extend kernel height beyond 3, two computational passes are required, with activation and kernel reloading in-between. This scheme can be extended across more rows to support even larger kernels if necessary. However, 11x11x3 and 7x7x3 kernels of the first layers of AlexNet and ResNet, respectively, can be handled through careful input data layout without explicit large kernel support, as described below [27, 30].

3.3.2.4 Padding, Pooling and Stride

Padding across activation tensor height can be done easily by adjusting which input slices are loaded into which columns. For example, with padding = 1, when computing the first output slice, one of the columns will not be used since it spatially corresponds to the padded-out input region, as shown in Figure 3.7 b). To perform padding across the width, we introduce

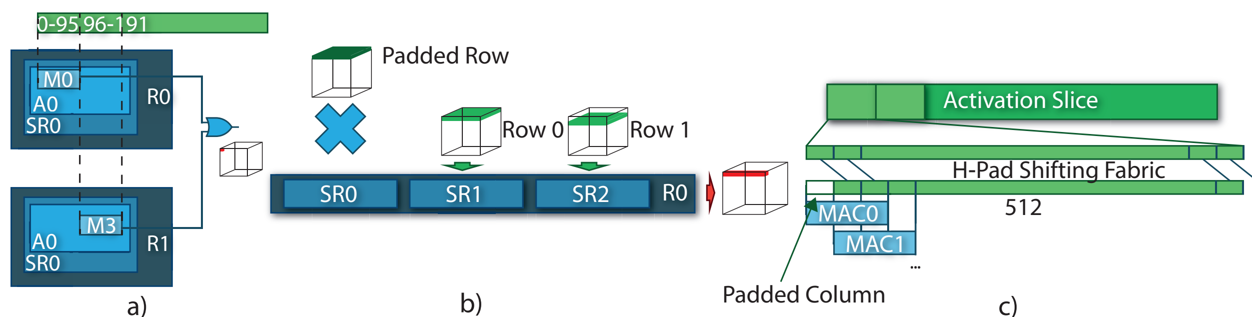


Figure 3.7: Extending kernel size up to 6x6 by coupling adjacent rows (a), enabling padding through row scheduling for height (b) and configurable shifting fabric before array inputs for width (c).

an additional *horizontal padding shifting fabric*, that can shift inputs in increments of 32 to the left before passing them into the columns, which is shown in Figure 3.7 c). The overhead of the shifting fabric is minimal, as it is shared between all rows.

For pooling across output height, ACOUSTIC exploits computation skipping by using proportionally shorter streams and not resetting output counters between successive computation phases. As a result, outputs that fall under the same pooling window are averaged together using the scaled addition property of stochastic stream concatenation. It means each compute pass is shortened proportionally to the pooling size.

Pooling across output width is done by output counters. Output counter with pooling support has small (2x-3x) parallel counters before them, which allows them to accumulate adjacent outputs that fall under the same pooling window. This choice allows us to shorten the streams proportionally to the pooling window width. As explained in Section 3.2.3, the area overhead of this solution is minimal. Stridden convolutions can be performed by reloading more than one column at a time, and dropping some of the results - for example, a 2x2 stride would require reloading two input slices between computation phases and dropping every other MAC result. Supporting arbitrary stride across the height does not incur any

extra area overhead. Across the width, however, using arbitrary stride would require a crossbar structure hence only limited stride window sizes are considered, e.g., 2x2. ACOUSTIC naturally prefers pooling as opposed to striding for dimensionality reduction.

3.3.2.5 First Layer Support

ACOUSTIC architecture is optimized for activation tensors with a depth in multiples of 32. It generally holds true for commonly employed models, except for the first layer [27, 1, 30]. Naively, a layer with a depth smaller than 32 can be implemented by storing just a single 1x1x3 input slice per every 32 values. Unfortunately, that would result in underutilization and slow processing of the first layer. Through a careful input data layout, however, good utilization of shallow layers can be achieved without any additional hardware overhead. Consider the first layer of AlexNet, which uses 11x11x3 kernels with a stride of 4. We can pack 4 1x3 partial slices per row, from two rows, into a single block of 32 values, resulting in the utilization of 2*4*3 = 24 out 32 values per block [27]. We can then pack the whole input width of 227 values across two rows into 56 such blocks and subsequent two rows into another 56 blocks, utilizing 2*56*24 = 2,688 values out of 4096 possible for a single column. A similar data packing method can be performed to support ResNet-style [30] 7x7x3 kernels with 2x2 stride in the first layer.

3.3.2.6 Fully-Connected Layer Support

ACOUSTIC supports fully-connected (FC) layers in the most straightforward manner possible. Since FC layers cannot re-use weights without employing batching, ACOUSTIC cannot capitalize on weight re-use within an array. This means that only a single MAC in a given array can be used. Because of that 416 out of 512 inputs to an array are wasted. However, if the fully-connected kernel is extended across six successive rows, their collective arrays can cover all 512 inputs with individual weights. The corresponding outputs of those rows

need to be then accumulated together, which is supported by the ACOUSTIC fabric. While this is highly unoptimized, and leads to 87.5% underutilization we argue that there is not much point in further optimizing the FC performance. While the first breakthrough CNN architectures like AlexNet and VGG relied on multiple, large FC layers with 10s of millions of weights each, newer ones like ResNet or Inception rely on a single, relatively small ones, which has very negligible impact on overall performance [27, 1, 30, 31].

3.3.2.7 Underutilization

Given the sheer number of multiply-accumulate units used in our architecture and its reliance on inflexible connectivity fabric, non-ideal resource utilization is unavoidable. We summarize the following main sources of underutilization:

- Filter Size - filter sizes other than 3x3 will cause compute resources to be underutilized.
- Input size - while arbitrary activation height is supported without underutilization, optimal width support is tied to the array/sub-row size. For example, most convolutional layers in ResNet will have a utilization of 87.5% due to their sizes not being a multiple of 16.
- Padding - padding will cause column underutilization when computing first and last output rows, resulting in utilization between 90.4% and 98.2%, depending on the activation tensor height.
- Stride - Underutilization caused by strided convolutions can be severe - a 2x2 stride means that every other MAC is not used. While it is supported, ACOUSTIC architecture is optimized towards exploiting computation skipping in pooling layers. It achieves both goals of strided convolution - dimensionality reduction and reduced computation without loss of accuracy [176].

We do not consider this a major issue of our architecture for two reasons. First, even

with 50% or lower utilization, the effective number of multiply accumulate units is still on the order of hundreds of thousands. Second, unused MACs and SNGs do not contribute to dynamic energy consumption - because their input values are zeroes, AND-based multipliers perform operand gating, removing any switching propagation.

3.3.3 Control

For any convolutional or FC layer, ACOUSTIC reloads activation and weight buffers multiple times depending on the number of rows and kernels of activation and output tensors. While this dataflow can easily be directed by a centralized FSM-style control unit, we opted for a distributed control scheme. It allows us to keep individual control modules simple while enabling overlapping different phases to reduce overall latency. We have developed a restricted instruction set shown in Table 3.2. The main control unit is the *Dispatcher* which reads the instructions, distributes them to other control units, maintains execution loops, and enforces synchronization through barriers. Four types of execution loops are supported, each with its iteration counter:

- Kernel loops iterate through kernels in batches of size equal to the number of rows (32).
- Batch loops iterate through multiple images for each layer. Batching should only be used when all activations can be stored on chip, as ACOUSTIC does not reload activations from external memory during processing.
- Row loops iterate through layer output height dimension.
- Pooling loops iterate through pooling window height dimension.

Each loop has its dedicated registers, and only one of each type can be used at a time. Loops themselves can modify other instructions through base addresses and increments. For

Table 3.2: ACOUSTIC control modules and their respective instructions.

Module	Instruction	Description
DMA	ACTLD/ST	Load/store activations from/to external memory
	WGTLD	Load weights from external memory
MAC	MAC	Compute
ACTRNG	ACTRNG	Load activations into SNG buffers
WGTRNG	WGTRNG	Load weights into SNG buffers
	WGTSHIFT	Shift weight SNG buffers
CNT	CNTLD/ST	Load/store activations from/to output counters
DISPATCH	FOR*/END* K/B/R/P	Kernel/batch/row/pooling loop
	BARR	Barrier

example, the kernel loop provides a base address and increment for loading weight buffers (WGTRNG). The dispatcher will modify and send instructions to other control units, which can be implemented as simple counter-based FSMs. Each of them will maintain a small FIFO to buffer multiple instructions and output an IDLE signal to the dispatcher once all instructions are processed. Instructions will continue to be dispatched until a barrier is encountered. Once that happens, the barrier mask will be compared with combined IDLE signals to determine if the execution can continue. This behavior allows ACOUSTIC to run multiple operations concurrently, e.g., loading weights for the next layer while processing the current one.

Overall, ACOUSTIC supports an extensive number of operations which allows it to implement the majority of image recognition models. Convolutions with different kernel, padding, and pooling sizes, fully connected layers, ReLU activations, and residual connections are all supported.

3.3.4 Evaluated ACOUSTIC Architectures

In this section, we parametrize the ACOUSTIC architecture to two reasonable choices that we evaluate in the next section. Besides the compute engine size, three main factors affect ACOUSTIC's performance: clock frequency, off-chip memory bandwidth, and on-chip memory size. Increasing the clock frequency speeds up the computation, but may require higher memory bandwidth not to be memory bound. Figure 3.8 shows that for the bandwidth achievable using different DDR3 standards, latency becomes memory limited at around 300 MHz or below. For smaller layers, that ”boundary” frequency will be much higher (e.g., above 1 GHz for 128 3x3x128 kernels). Further, for ultra-low energy accelerators, support for large model sizes may be unnecessary, and therefore the support for DRAM can be omitted. Finally, activation memory can be sized up to support larger batch sizes.

We evaluate two versions of ACOUSTIC architecture - low power (LP) and ultra-low-power (ULP). Performance estimation for both configurations was done using a commercial

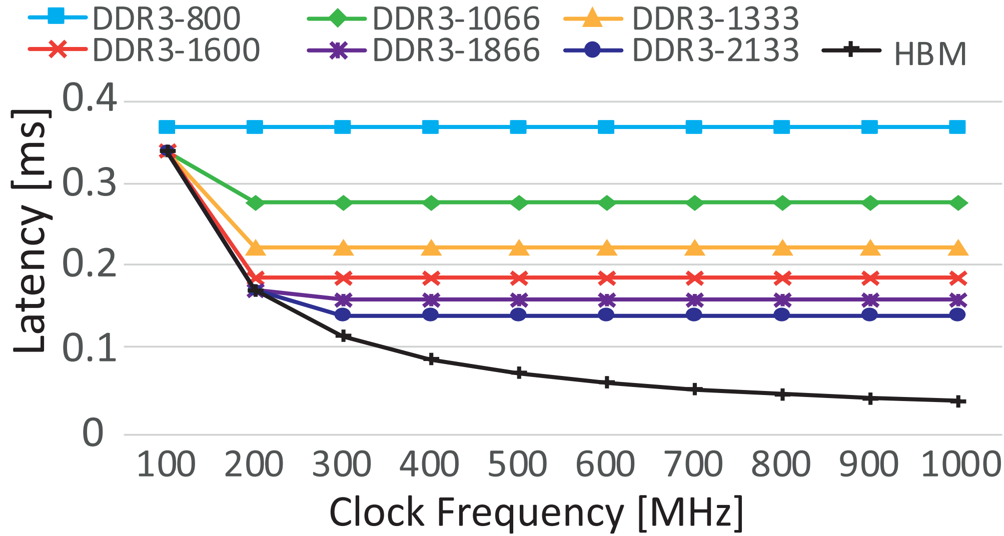


Figure 3.8: Latency of processing a convolutional layer with 16x16x512 inputs and 512 3x3x512 kernels and pre-loading 512 3x3x512 kernels for the subsequent layers using different clock frequency and external memory interfaces, using temporarily-unrolled 256-long split-unipolar streams.

28nm library. The LP variant (details in Table 3.4) is intended to be integrated in mobile SoCs with limited area and power budgets. It has enough on-chip weight memory (147.5KB) to store all weights for commonly encountered convolutional layers. For large fully-connected layers, a new batch of weights is fetched while the current one is being processed. It has enough on-chip activation memory (600KB) to process the most commonly used CNNs without ever having to offload activations off-chip [27, 1, 31, 30]. In cases where that is not possible, outputs are offloaded to external memory and fetched back when necessary for the next layer, which is supported by ACOUSTIC ISA. The ULP variant (Table 3.5) targets low-complexity inference (e.g., MNIST digit recognition using LeNet-5) on extremely resource-constrained devices. It is meant to compete with the analog, and neuromorphic approaches in terms of energy efficiency [6]. It has 2KB of activation memory and 3KB of weight memory.

3.4 Evaluation & Results

3.4.1 Evaluation Methodology

SC is extremely slow to accurately simulate in software, mainly because of randomization [166]. To aid in computationally tractable design space exploration, we opted to decouple functional and performance simulations. For any trained neural network model, accuracy is evaluated using our custom SC functional simulator, which models just the computation part using bitstreams. It is given the network model, test dataset, trained weights, and SC configuration, e.g., stream lengths, RNG scheme, which it uses to compute test accuracy and compare against training results. The same configuration and neural network model (described in ACOUSTIC ISA) is then fed to the custom performance simulator, whose goal is to accurately model execution time and data movement without simulating the actual computation. The performance simulator is also fed power, area, and latency numbers for individual system components, which it uses to generate accurate processing energy and latency numbers. We used the TSMC 28nm library with Synopsys Design Compiler synthesis tool to obtain area, latency, and power numbers for the MAC array, buffers, SNGs, and counter/ReLU units. Memory, both SRAM and DRAM, were modeled using CACTI 6.5 [177].

We use Eyeriss as a baseline for the LP variant, which is a template for most spatial accelerators [2, 13, 130]. To model latency and energy consumption, we use the simulator presented in [178]. We compare our numbers to original Eyeriss configuration with 168 processing elements (PEs), as well as a scaled-up version with 1024 PEs, both scaled to 28nm technology and 8-bit precision. Where possible, we also compare to SCOPE, a state-of-the-art SC neural network accelerator [166]. SCOPE is a flexible DRAM-based in-memory compute accelerator, with only multiplication performed in the stochastic domain. SCOPE numbers are reproduced from [133, 166] and scaled to 28nm. For the ULP variant, we compare with MDL-CNN [6], a time-based convolution engine with a similar area footprint

and Conv-RAM [7], analog, in-memory convolutional engine, both scaled to 28nm. All ACOUSTIC configurations use split-unipolar representation with 2x128-bitstreams.

3.4.2 ACOUSTIC Accuracy

Accuracy results are shown in Table 3.3. AlexNet on ImageNet is too large for our current SC simulator, so SC validation accuracy is not available [1]. We do stochastic stream-based training with an approximate OR, as described in Section 3.2, to show the achievable accuracy of ACOUSTIC. As Table 3.3 shows that ACOUSTIC accuracy is the same as SCOPE, and it can achieve an accuracy similar to 8-bit fixed point hardware with stream lengths of 512 (i.e., 256x2 for split-unipolar). We believe part of the remaining gap is due to the use of approximate OR during training, and better but computationally tractable approximations are part of our ongoing work.

Table 3.3: Accuracy comparisons.

			Validation Accuracy [%]		
Network	Dataset	Stream Length	8-bit Fixed Pt	SCOPE	ACOUSTIC
LeNet-5	MNIST	128	99.2	99.32	99.3
CNN	SVHN	256	90.29	N/A	86.75
		512		N/A	89.02
	CIFAR-10	256	79.9	N/A	74.9
		512		N/A	78.04

[1]SCOPE [166] only reports results on MNIST.

3.4.3 Runtime Configurable Precision

Second, each counter has a configurable shift module at the output, which can scale results up and down in the power of 2's, so they are consistent with the "native" length of 128 bits. ACOUSTIC can support runtime-configurable stream lengths from 32 to 1024 (in steps of powers of 2). Further, thanks to its programmable nature, stream lengths can be configured on a layer-by-layer basis, or even at a finer granularity, at a cost to the code size. Different stream lengths for different layers enable fine-grained control of accuracy and performance. Figure 3.9 shows achievable accuracy and relative latency increase for different stream length configurations on the CIFAR-10 CNN, running on ACOUSTIC ULP. △ represents stream length of 256 in the 1st layer, while □ represents 512 and ◊ represents 1024. Blue represents a stream length of 256 in the 2nd and 3rd layer, while green represents 512 and red represents 1024. The stream length of the 4th layer is also modified, and longer stream lengths increase accuracy and latency. By slowing the accelerator down two times, the accuracy improvement of 4.7% can be achieved. In theory, quadrupling the stream lengths in all layers should result in a 4x latency increase. The much lower actual increase is caused by the small size of the compute engine in the ULP version. Full slices in layers 2 and 3 cannot fully fit in activation SNG buffers, resulting in frequent buffer reloading, and lowering the computation duty cycle. An analysis of the relative precision importance of each layer can also be obtained from those results.

3.4.4 Area & Power Breakdown

Area breakdowns for ACOUSTIC LP and ULP configurations are shown in Figures 3.10 a) and b), respectively, while power breakdowns are shown in Figures 3.10 c) and d). As can be seen, MAC arrays are significant contributors to both area and power on the ACOUSTIC LP variant. Weight buffers, while being significant contributors to the area, have much lower relative power consumption due to very infrequent switching. A more area-efficient

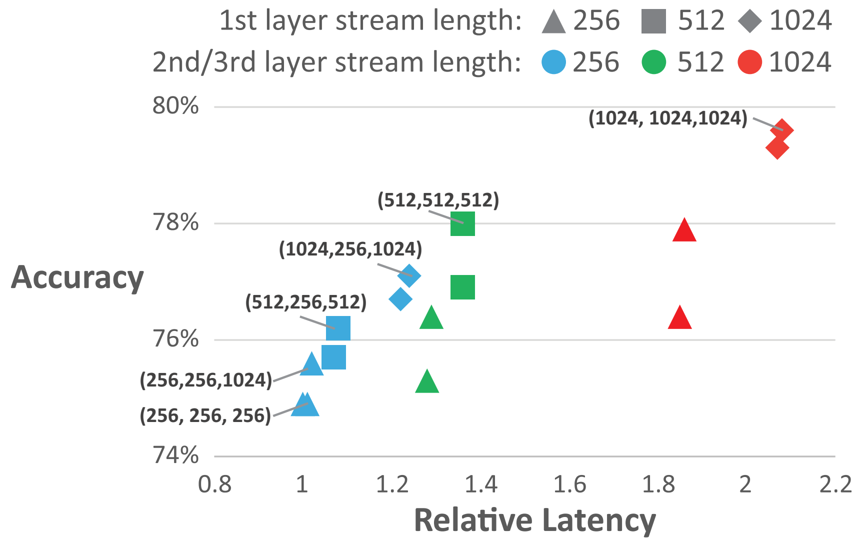


Figure 3.9: Accuracy and performance at different stream lengths for the CIFAR-10 CNN on ACOUSTIC ULP. Labeled points are pareto points with the numbers representing stream lengths in 1st layer, 2nd & 3rd layer, and 4th layer.

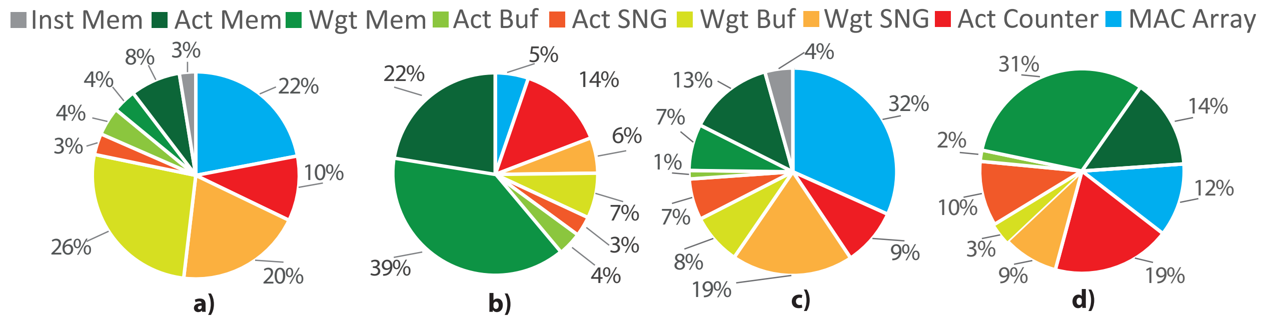


Figure 3.10: Area breakdown for ACOUSTIC LP (a) and ULP (b) and power breakdown for ACOUSTIC LP (c) and ULP (d).

implementation of weight buffers should, therefore, be explored. The area and energy of the ULP variant are dominated by activation and weight memories.

3.4.5 Performance Comparisons

Table 3.4 compares area, power, and clock frequency for all accelerators, as well as inference throughput (frames/s) and energy efficiency (frames/J) for different models. ACOUSTIC outperforms conventional fixed-point accelerators. LP variant can be as much as 38.7x more energy-efficient than Eyeriss with 1k PEs, depending on the network model. It is also more energy-efficient than SCOPE - up to 79.6x higher frames per Joule. SCOPE requires hundreds of mm^2 of area, which makes it unsuitable for edge inference. SCOPE multiplies stochastic streams in parallel, instead of using streaming like ACOUSTIC. This mandates using multiple, large DRAM arrays to achieve low latency.

The latency of AlexNet and VGG is primarily dominated by fully-connected layers. Both VGG and AlexNet have multiple, large FC layers, with tens of MB of weights, and ACOUSTIC is not optimized toward those. On the Resnet-18 model, which has only a single, small FC layer, ACOUSTIC delivers lower latency than for AlexNet, despite Resnet-18 being ≈2x more computationally intensive.

Table 3.5 shows that the ACOUSTIC ULP variant, with a comparable area footprint, can deliver up to 123x speedup over MDL-CNN, and is 1.33x more energy efficient[2]. It has 8.2X higher throughput than Conv-RAM with similar energy efficiency. Furthermore, ACOUSTIC uses 8-bit precision for both weights and activations, while both MDL-CNN and Conv-RAM use binarized weights, resulting in a 1%-3% drop in accuracy for MNIST. For a fair comparison, we are using 128-long bitstreams for ACOUSTIC ULP and non-accelerated MDL, such that neither architecture sacrifices any accuracy.

---

[2]Only convolutional layers are compared as CONV-RAM, and MDL-CNN do not report performance on FC layers.)

Table 3.4: Performance comparison between ACOUSTIC LP and other fixed-point and stochastic accelerators.

	Eyeriss 8-bit			ACOUSTIC
	Base	1k PEs	SCOPE	LP
Area[mm2]	3.7	15.2	273.0	12.0
Power[W]	0.12	0.45	N/A	0.35
Clock[MHz]	200	200	125	200
AlexNet Fr/J	306.9	381.2	136.2	2590.6
Fr/s	41.1	210.7	5771.7	238.5
VGG-16 Fr/J	14.4	18.7	9.1	723.8
Fr/s	1.8	8.4	755.9	93.2
Resnet-18 Fr/J	295.6	380.3	N/A	2471.6
Fr/s	34.0	182.5	N/A	542.6
CIFAR-10 CNN Fr/J	N/A	N/A	N/A	1.01M
Fr/s	N/A	N/A	N/A	46,168

Table 3.5: Performance comparison between ACOUSTIC ULP, MDL CNN [6] and Conv-RAM [7] on convolutional layers of LeNet-5 and CIFAR-10 CNN.

	Conv-RAM	MDL CNN	ACOUSTIC ULP
Domain	Analog	Time	SC
Precision [A/W]	6b/1b	8b/1b	8b/8b SC
Area [mm2]	0.02	0.124	0.18
Power [mW]	0.016	0.03	3
Clock [MHz]	364	24	200
LeNet-5 Fr/J	40M	33.6M	41.7M
Fr/s	15,200	1009	125,000
CIFAR-10 CNN Fr/J	N/A	N/A	697K
Fr/s	N/A	N/A	2100

3.5 FPGA Evaluation

To prove the feasibility of the ACOUSTIC concept, we present a functional FPGA implementation. We used the AVNET Ultra96-V2 board, with a Xilinx Zynq UltraScale+ MPSoC ZU3EG device. It has a built-in ARM-based SoC which we used to interface with the ACOSUTIC accelerator. We implemented the ULP version of ACOUSTIC in Verilog RTL and synthesized and implemented it using Xilinx Vivado 2019.2 software. The current RTL version does not yet include a DMA engine, so we have upsized the on-chip memory sizes compared to the VLSI ULP version, to be able to fully run LeNET and CIFAR-10 CNN networks. We used an accurate cycle counter to measure execution latency.

Table 3.6 shows the hardware utilization and peak performance for ACOUSTIC ULP and certain fixed-point FPGA convolutional neural network accelerators. For performance,

we report giga-MACs per second (GMACS), to avoid the ambiguity of GOPS [3]. First, notice that ACOUSTIC uses fewer flip-flops than any other architecture, due to the combinational nature of SC compute. Flip-flop usage has crucial implications for VLSI implementations, as sequential elements can occupy large portions of the chip. Second, while there are accelerators that use fewer logic resources, they all rely on large numbers of costly DSP resources. The only accelerators that can outperform ACOUSTIC in terms of GMACS require an order of hundreds of those units, which restricts them to very large and expensive FPGA devices. ACOUSTIC, in contrast, requires only logic and memory resources, meaning it can easily scale to the underlying hardware. While other accelerator's performance is tied to their chosen number representation, ACOUSTIC can support different precision, and performance, at runtime, without any hardware reconfiguration.

Using the hardware cycle counters, we have empirically verified the simulated LeNet-5 and CIFAR-10 CNN throughputs listed in Table 3.5, corrected for clock frequency. Our hardware simulator results are within 4% and 6% on LeNet-5 and CIFAR-10 CNN, respectively. Discrepancies are mainly due to the lack of an actual DMA in our current hardware, and certain dead cycles caused by inefficiencies in the current RTL implementation of the control scheme.

3.6 Demonstration Chip

To further demonstrate the potential of the ACOUSTIC architecture, we have tapeout out a chip using Global Foundries' 14LPP technology node. The taped-out chip is based on the ULP/FPGA variant, with minor modifications described below.

[3] Where it was not clearly stated, we assumed authors reported their numbers assuming OP=MAC, to their benefit.

Table 3.6: FPGA utilization and performance comparison between ACOUSTIC ULP and other convolutional neural network accelerators. ACOUSTIC performance is for stream lengths in range of 32 to 256-bits.

Architecture	Device	FF [k]	LUT [k]	BRAM	DSP	Clock [MHz]	GMACS
SC DEMO	MPSoC ZU3EG	26.3	46.5	38	0	140	84-672
ANGEL-EYE [179] 4PE 16CONV	XC7Z045	34.1	43.1	203	400	150	105.2
Aristotle [180]	ZYNQ 7030	34.1	43.1	203	400	150	172.8
CNNA [181]	ZU9EG	28	95	900	1200	250	500
NullHop [125]	ZYNQ7100	229	107	386	128	60	8.6
DCNN 2 MACs[182]	ZYNQ 7045	N/A	112.5	131.5	392	145.45	41.5

3.6.1 Architecture

The block diagram of the taped-out accelerator is shown in Figure 3.11. To further emphasize the benefits of using stochastic computing, we highlight components that support variable precision in blue and reuse-oriented design choices in red. Numbers associated with each technique refer to the order used in Figure 3.14. The chip consists of control logic, an activation scratchpad coupled with buffers and stochastic number generators (SNGs), and six multiply-accumulate (MAC) block rows, each with its weight memory (total 131KB), and output counters, pooling, and activation logic. Control consists of instruction memory (4KB), a dispatcher, and distributed control units. The activation scratchpad is organized as a ping-pong buffer (2 banks of 16KB). After conversion into stochastic streams, *activation broadcast* is used across all six MAC rows, enabling a high level of spatial reuse. All MAC rows use the same activations, but each can compute a single output channel in a convolutional layer, or three rows can be coupled to compute one output channel in a fully-connected layer.

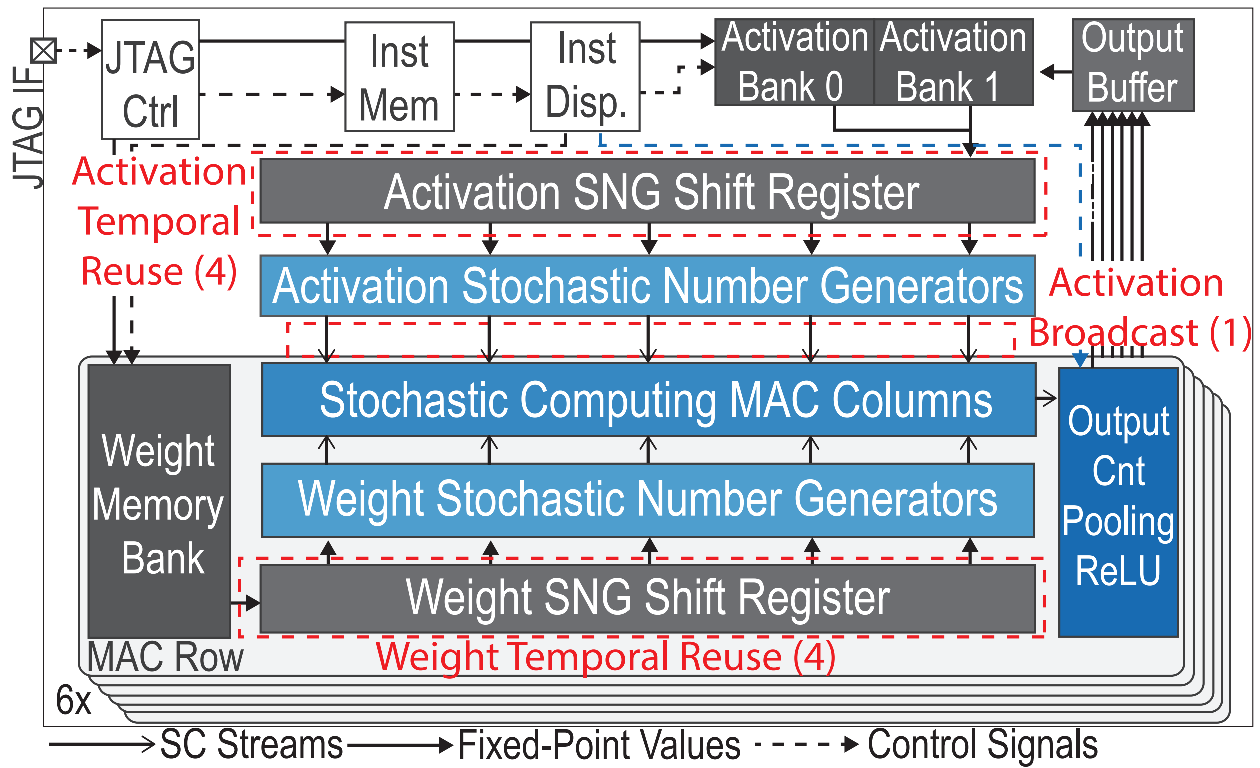


Figure 3.11: Overall accelerator architecture.

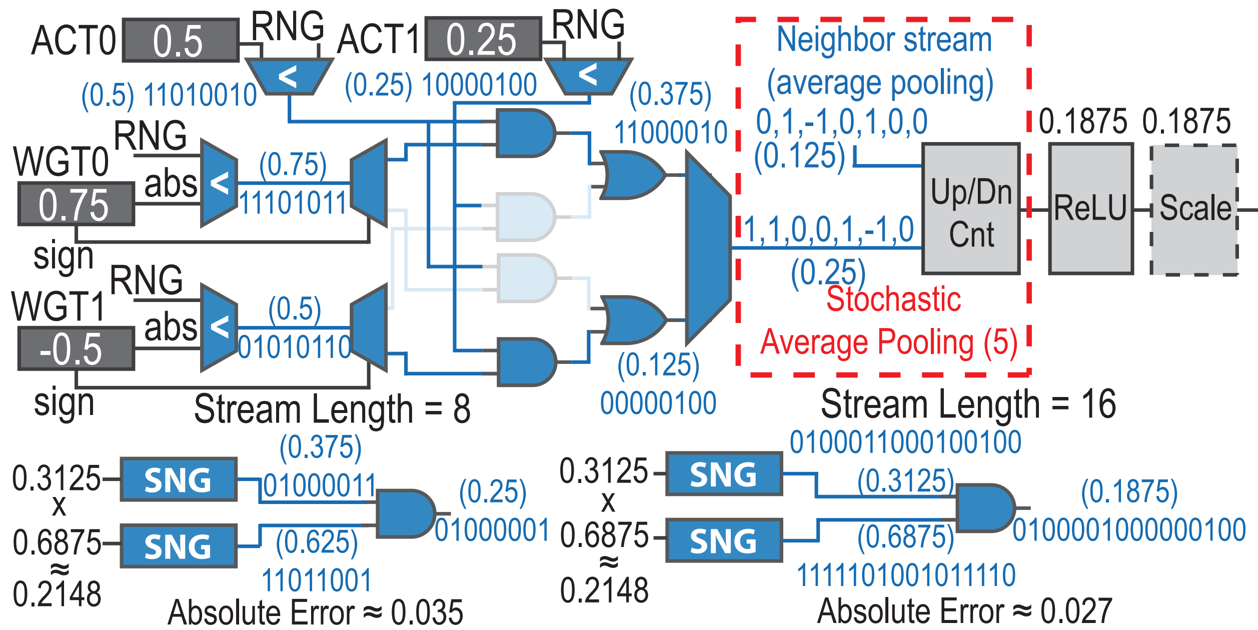


Figure 3.12: SC split-unipolar MAC and stochastic average pooling (top). Precision-latency trade-off using different stream lengths (bottom).

3.6.2 SC Computation

Figure 3.12 explains how the underlying SC computation is performed by the taped-out chip, through an example of a 2-way dot product using 8-bit long streams. We use comparator-based SNGs, with 7-bit linear feedback shift registers [5]. Multipliers and adders are implemented using AND and OR gates, respectively. Their small size allows us to pack extremely wide dot products, up to 200-wide, without timing or congestion issues present in fixed-point designs. After accumulation, positive and negative streams are subtracted and fed into a counter for conversion to fixed-point. Each up/down counter accepts outputs of two neighboring accumulation trees to implement X-dimension (*horizontal*, e.g. 1x2) part of *stochastic average pooling* by concatenating the streams corresponding to two adjacent outputs in the same output row. Y-dimension (*vertical*, e.g. 2x1) part of pooling is implemented sequentially, using the previously described sliding window. Pooling is only used after convolutional layers. Counters are followed by ReLU activation and a configurable power-of-2 scaling unit to handle different stream lengths. Scaling factors range from 1/4 to 32, are programmable, and can be set on a layer basis. Figure 3.12 also shows how by changing the stream length, the computation precision can be changed at runtime.

3.6.3 Computation Mapping

Convolution mapping is shown in Figure 3.13. Each MAC block row is organized into five columns, each corresponding to one input and filter row across 4 or 8 input channels. Fixed-point input and weight rows are loaded into the SNG buffers of their respective column. This way, a complete 5x5 filter can be computed in parallel, generating one complete row of outputs per MAC block row. Both sets of SNG buffers are organized as shift registers, meaning that after one iteration, a new input row is shifted in, and the subsequent output row can be computed. This *sliding activation reuse* behavior emulates a vertically-sliding convolutional window, reducing the number of required memory accesses.

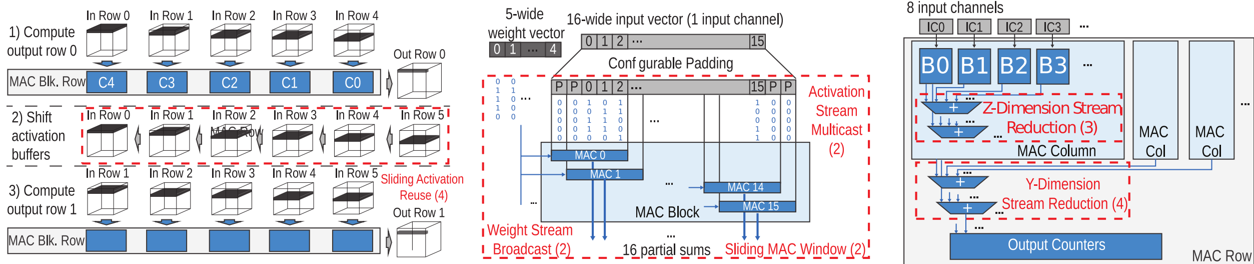


Figure 3.13: Mapping of convolutional layers in MAC rows, and memory/stream generation amortization through data reuse. Shift-register organization emulating sliding window (left), MAC block with input/weight reuse and padding support (center), block organization implementing different levels of reduction (right).

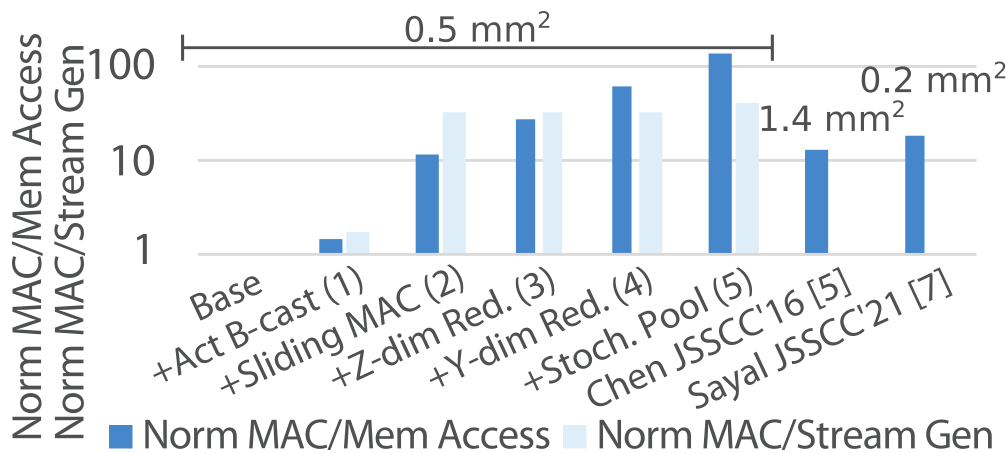


Figure 3.14: Normalized ratio of MAC to memory accesses and stream generations compared to fixed-point designs. Accelerator area scaled to 14nm is included.

Each column is organized into eight SC MAC blocks, each corresponding to one input channel, operating in a fully-streaming manner. A block takes 16 inputs and 5 weights and implements a *sliding MAC window*, generating up to 16 partial output streams depending on padding. *Activation stream multicast* with overlap is used between successive dot-products, and *weight stream broadcast* is used for all dot products in a block, amortizing stream generation cost. Every two neighboring blocks can be coupled to support wider input rows. Corresponding output streams from all 8 blocks are then reduced (*Z-dimension stream reduction*), after which corresponding outputs from all 5 columns are reduced (*Y-dimension stream reduction*). This hierarchical, SC reduction enables wide dot products (up to 200-wide) to be unrolled spatially, amortizing the cost of converting stochastic streams to fixed-point outputs. Our reuse-oriented design choices make it possible to perform over 130 MACs per memory access, and over 40 MACs per stream generation, as shown in Figure 3.14, lowering the relative energy cost of those operations. This high level of data reuse was only possible by using very dense SC computation – fixed-point designs have an order of magnitude lower memory access reuse.

Since computation, highlighted in blue, operates purely on stochastic streams, it can support arbitrary precision by adjusting SC stream length, only limited by the width of the output counters. Precision selection is possible on sub-layer granularity, e.g., groups of filters. Our custom instruction set enables selective gating of MAC block rows, columns, and blocks to support smaller activation or filter sizes. Filters or inputs that cannot be fully unrolled using existing resources are computed sequentially. A set of hardware loops with multi-stride memory accesses enables a variety of layer shapes and dataflow choices. Our accelerator can therefore support end-to-end inference with different types and shapes of layers.

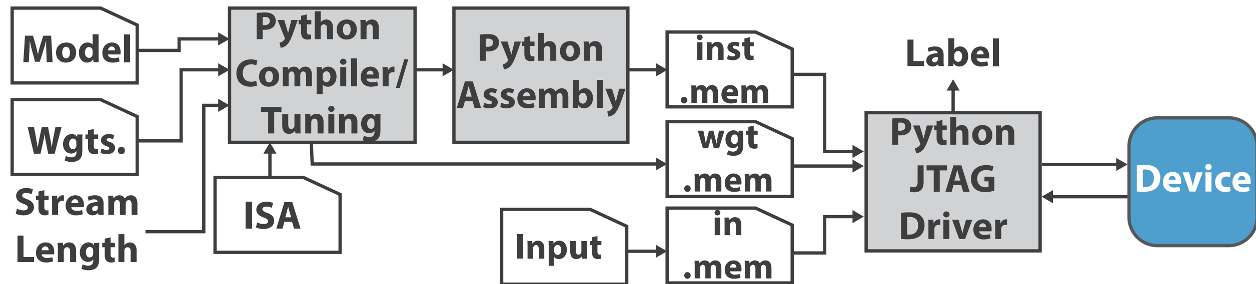


Figure 3.15: Model deployment pipeline.

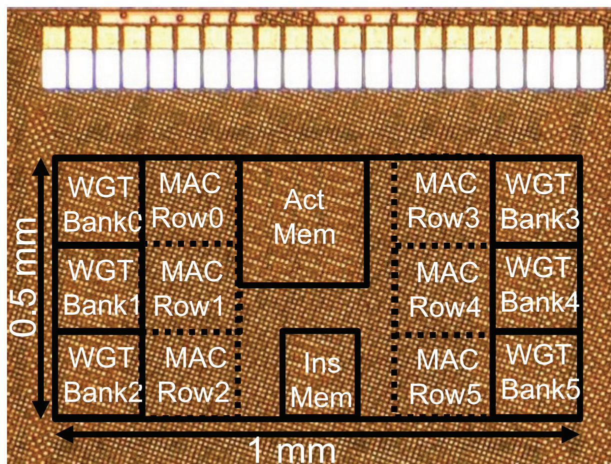
3.6.4 Evaluation & Results

To speed up the training procedure, models are first trained using a fast generation scheme that maximizes stream sharing and then fine-tuned using an accurate model of the hardware. The effect of OR accumulation is modeled using an additional activation function described earlier. Models are trained on MNIST, CIFAR-10, and SVHN datasets, using models shown in Table 3.7. An output counter overflow issue in the taped-out design causes the deployed models to use aggressive scaling factors, which result in a 0.2-7.4 p.p. accuracy drop compared to simulated results without the overflow. The scaling factor is set to be $2^{\lceil log_2(max(|a|))\rceil}$, where $max(|a|)$ is the largest magnitude observed in the output activations of a layer. This scaling factor ensures that output activations do not overflow after scaling, and that the scaling function can be achieved using simple shifts. Accuracy could also be improved through larger networks, or more recent SC accuracy improving techniques, such as the ones proposed in [64]. Once trained, models are translated into a sequence of accelerator instructions using a compiler under active development, which is then programmed onto the device, as shown in Figure 3.15.

The chip, shown in Figure 3.16, was fabricated in 14nm LPP technology. The core of our accelerator occupies a 0.5 mm^2 area and operates at 0.6-0.9V voltage and a maximum frequency of 500MHz. Figure 3.17 shows the accuracy, latency, and energy measurements

Table 3.7: Datasets and models used in evaluation. Model sizes are limited by available on-chip memory.

Dataset	Model	Model Architecture	Size [kB]
CIFAR-10,	tinyConv	CN5x32-CN5x32-CN5x64-FC10-BN*	89
SVHN	tinyConv-L*	CN5x32-CN5x32-CN5x32-CN5x32-FC10-BN*	81
MNIST	LeNet-5	CN5x6-CN5x16-FC120-FC84-FC10	62
	LeNet-3	CN5x6-CN5x16-FC10	7



	Specifications
Technology	14nm LPP
Die Area	5 mm²
Core Area	0.5 mm²
Supply Voltage	0.6-0.9V
Frequency	250-500 MHz
Precision	8-64 bitstreams
Memory	167 KB
Power	16 mW @ 250 MHz, 0.6V
	68 mW @ 500 MHz, 0.9V

Figure 3.16: Die shot and specifications.

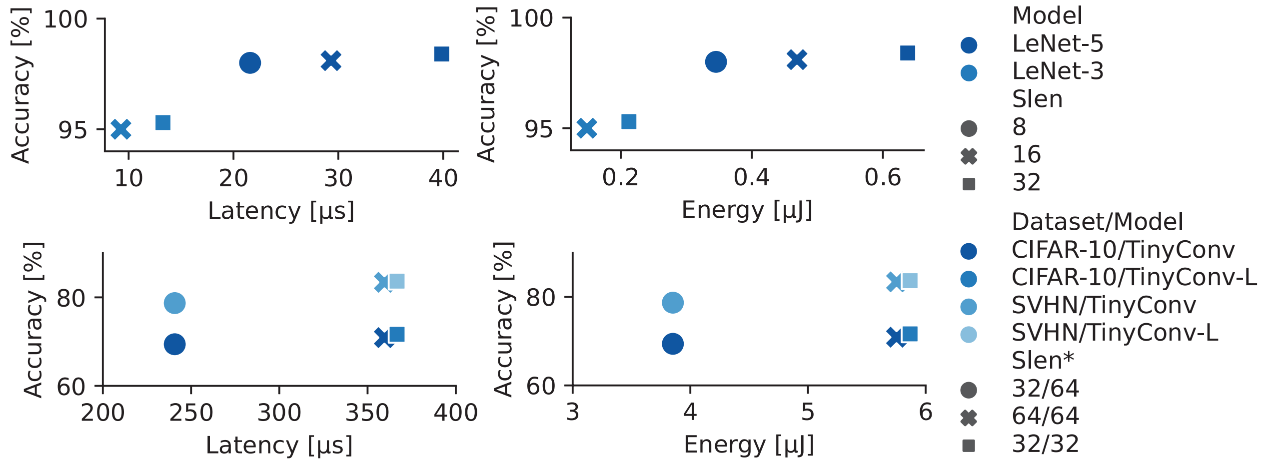


Figure 3.17: Accuracy, latency (left), and energy (right) on the MNIST (top), SVHN and CIFAR-10 (bottom) datasets.

on different networks and datasets. We highlight that SC exposes a precision-performance tuning knob, by adjusting the stream length. For example, on the MNIST dataset, changing the stream length from 32 to 16 reduces the accuracy by only up to 0.3 p.p., while reducing energy and latency up to 31%. On SVHN, accuracy can be improved by up to 4.7 p.p. with a 50% increase in latency and energy. Non-ideal performance scaling is caused by control logic overheads. Thanks to highly compact SC compute we achieve higher data reuse than other accelerators, resulting in lower relative memory power contribution, as shown in Figure 3.18.

Peak energy efficiency ranges between 9.4 and 75 TOPS/W at 250MHz/0.6V, as shown in Figure 3.19. Table 3.8 shows a comparison with prior work. Overall, SC offers unparalleled computational density in terms of MAC units per mm^2. Our chip outperforms fixed-point accelerators [183] in energy efficiency while enabling configurable precision. While bit-serial architectures [121] can have high peak energy efficiency, it is only achievable at single-bit precision. Mixed-signal, binary design in [58], achieves extremely high energy efficiency, but it comes at a cost of little configurability, supporting only one convolutuional filter size.

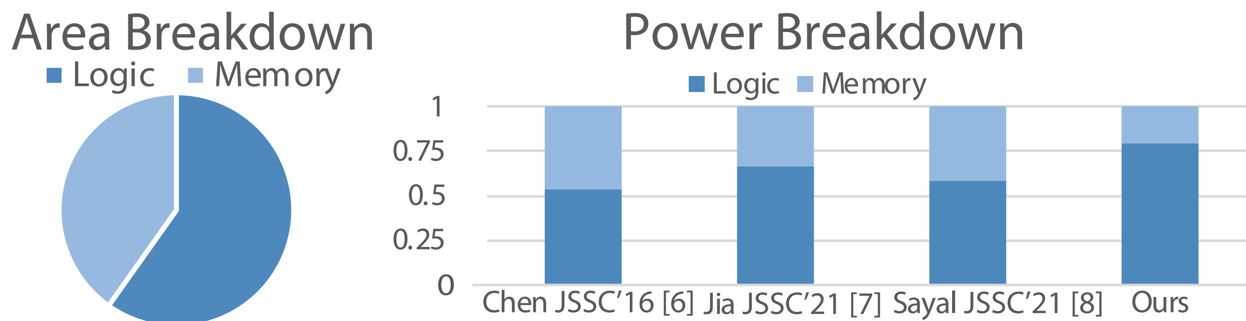


Figure 3.18: Area (left) and power (right) breakdown, compared to [2, 3, 4].

It requires using much larger, slower models to improve accuracy. We also outperform, or approach the efficiency of neuromorphic and analog designs [184, 6], without suffering from their scalability and programmability issues, on account of being purely digital.

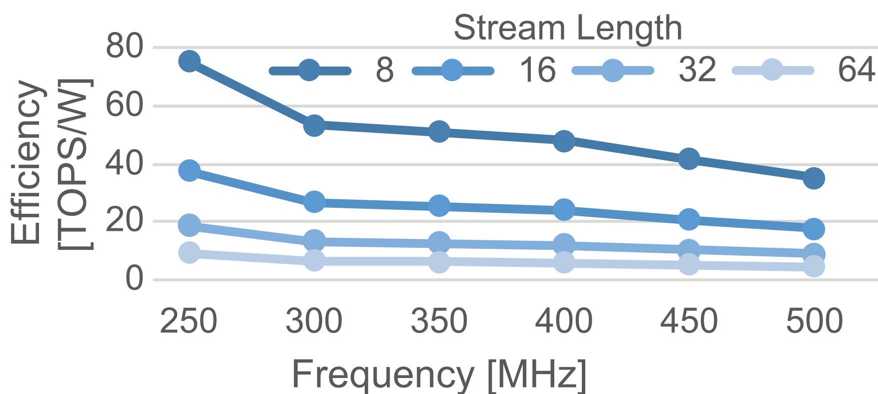


Figure 3.19: Peak energy efficiency at different stream lengths.

3.7 Related Work

3.7.1 Deep Learning using Stochastic Computing

Since SC allows very cheap implementation of multiplication and addition and neural networks are shown to be highly error-tolerant [185], SC has been used to accelerate neural

Table 3.8: Comparison table.

	ISSCC'21 [183]	ISSCC'18 [121]	JSSC'18 [58]	ISSCC'19 [184]	ISSCC'19 [6]	This Work
Type	Digital	Digital, bit-serial	Mix.-sig., binary	Neuro-morphic	Time-Domain	Digital, SC
Purpose	Train./Infer.	Infer.	Infer.[4]	Train./Infer.	Infer.[4]	Infer.
Node	7nm	65nm	28nm	65nm	40nm	14nm
Area [mm^2]	19.6	16	4.6	17.6	0.124	0.5
Voltage [V]	0.55-0.75	0.63-1.1	0.53-0.8	0.8	0.375-1.1	0.6-0.9
Clock [MHz]	1-1.6	200	1.5-10	20	0.2-3.1	250-500
Power [mW]	—	3.2-297	0.09-0.9	23.1-23.6	0.03	16-68
#MAC	32k[1]	3.5k[1]13.8k[2]	65k	—	—	19.2k
MAC/mm2	1.6k[1]	0.2k[1]0.9k[2]	14.2k	—	—	38.4k
Mem.	8MB	256KB	328KB	—	—	167KB
Precision	FP8-32,INT4/2	1-16	1	—	—	4-8[5]
TOPS/W	8.9-16.5[1]	11.6[1]50.6[2]	532-772[2]	3.42	12.08	4.4-75
TOPS/mm2	3.27-5.22[1]	0.086[1]0.46[2]	0.015-0.1[2]	—	—	0.3-4.8 (0.75-12)[6]
GOPS	62K-102.4K[1]	1.4K[1]7.3[2]	72-532[2]	—	0.365	150-2.4k
MNIST Perf.	-	-	-	97.8%, 3.4 TOPS/W, 100 kFr/s	97%, 4.65-12.08 TOPS/W[4]	95.1-98.7%, 1.1-9.5 TOPS/W, 25-215 kFr/s
CIFAR-10 Perf.	-	-	85.7-86.1% 532-772 TOPS/W, 0.04-0.24 kFr/s	-	-	69.4-71.7%, 2-6.3 TOPS/W, 2.8-8.3 kFr/s

[1] Int4. [2] Binary. [3] Fixed-function, MNIST only. [4] Convolutional layers only. [5] Effective precision with 8-64 bit SC streams and SC average pooling. [6] Without on-chip memory.

network inference. In most cases, SC is used to accelerate multiplication, and addition is performed using counters[106, 108, 171, 186, 104]. Those approaches have recently moved towards using deterministic, instead of pseudo-random sequences to generate stochastic streams [187, 110]. We plan to explore the applicability of such sequences in the ACOUSTIC framework as a part of our future work. [111, 188] used OR for addition for small networks and partially with rest of accumulation still handled by counters. MUX is also used to implement addition in SC, but they are either limited to one single layer[172] or show poor [189] or even no accuracy [190]. Apart from bipolar and unipolar stream, other representation methods like extended stochastic logic (ESL) [186, 191], two-line representation [192] and integer stream [193] are also proposed. Among them, ESL suffers from increased error due to having a stochastic stream in the denominator, while the other two suffer from increased hardware cost for computation. Prior FPGA implementations of SC-based acceleration have been shown, but those are fixed-function accelerators tailored towards a single network [194, 191], whereas ACOUSTIC is a scalable and programmable architecture.

3.7.2 Approximate and Programmable Precision Accelerators

Long-proven neural network resistance to approximation has become one of the primary means of reducing model size and complexity [168, 37, 169]. Binarization [47, 53], though resource-efficient, can suffer from accuracy problems and does not offer any flexibility in terms of computation precision. Bit-serial and bit-pair encoding accelerators [19, 195, 196, 197, 121, 198] decompose multi-bit operations into serially processed binary ones, therefore, allowing for accuracy vs. energy and latency tradeoffs similar to SC but are nowhere as compact as SC. Performance benefits here rely mainly on reducing the precision of computation below 8-bits, whereas we show ACOUSTIC being competitive at close to iso-8-bit precision. An in-memory version of bit-serial processing has also been explored [197, 133] but require modifications to heavily optimized memory fabrication process and large arrays to deliver performance, making them unsuitable for edge devices. Using dynamic voltage-frequency

scaling for energy-accuracy tradeoff has also been explored [199, 200]. Due to its inherent fault-tolerance, SC is also extremely well-suited to it [201], and we plan to explore its use in ACOUSTIC in the future.

3.8 Conclusion

In this paper, we presented the ACOUSTIC accelerator for convolutional neural networks. ACOUSTIC incorporates multiple algorithm optimizations: split-unipolar representation, stochastic average pooling with computation skipping, and training and hardware cooptimization. ACOUSTIC accelerator architecture is built around the idea of density-enabled data reuse, which allows it to significantly reduce the number of on- and off-chip memory accesses. ACOUSTIC architecture delivers server-class parallelism within a mobile area and power budget - a 12mm^2 accelerator can be as much as 38.7x more energy-efficient and 72.5x faster than conventional fixed-point accelerators. It can also be up to 79.6x more energy-efficient than state-of-the-art stochastic accelerators and can be implemented in order of magnitude less area than recent SC-based accelerators, delivering real-time performance in a mobile/IoT energy/area envelope. It can also take advantage of runtime-configurable precision to enable tradeoffs between latency and accuracy. Finally, we have shown a functional hardware implementation, proving the feasibility of our approach. Our ongoing primarily addresses developing fast training algorithms for ACOUSTIC to improve accuracy for large networks.

We have also presented the first stand-alone, configurable, and programmable SC NN accelerator in silicon. The chip, taped out in 14nm technology, achieves 2.4 TOPS and 75 TOPS/W throughput and energy efficiency. It has a custom ISA and supports end-to-end inference with convolutional and fully-connected layers of variable input and filter sizes. The use of SC makes it possible to pack 19,200 MAC units in a small area, enabling a high degree of spatial data reuse, amortizing the cost of SC conversion, and reducing the number

of memory accesses. By varying the stream length, it enables extensive accuracy-latency trade-offs.

CHAPTER 4

GEO - Pushing Stochastic Computing Further

ACOUSTIC, introduced in the previous Chapter, laid foundations for the design of stochastic computing neural network accelerators. However, it came with a host of issues, like noticeable accuracy loss, costly stream generation, and reliance on off- and on-chip memory bandwidth. In this Chapter, we propose GEO – Generation and Execution Optimized Stochastic Computing Accelerator for Neural Networks, which optimizes stream generation and execution components of SC, and bridges the accuracy gap between stochastic computing and fixed-point neural networks. It improves accuracy by coupling controlled stream sharing with training and balancing OR and binary accumulations. GEO further optimizes the SC execution through progressive shadow buffering and other architectural optimizations. GEO can improve accuracy compared to state-of-the-art SC by 2.2-4.0% points while being up to 4.4X faster and 5.3X more energy efficient. GEO eliminates the accuracy gap between SC and fixed-point architectures while delivering up to 5.6X higher throughput and 2.6X lower energy.

Collaborators:

- Tianmu Li, Electrical and Computer Engineering, UCLA.
- Professor Sudhakar Pamarti, Electrical and Computer Engineering, UCLA.
- Professor Puneet Gupta, Electrical and Computer Engineering, UCLA.

4.1 Introduction

As established in Chapter 3, stochastic computing has been enjoying a renaissance in deep learning acceleration for latency-, energy-, and cost-constrained devices [109, 166, 104, 110, 5]. Its approximate nature synergizes well with neural networks' inherent error-tolerant properties, enabling new axes of accuracy and performance tradeoffs [104, 202, 5]. However, precision remains the most significant barrier to wider SC adoption. Since the bulk of accuracy is lost in the addition operation, the majority of prior works opted for approximate parallel counter-based accumulation fabric [105, 203, 110, 202] or directly converting each multiplication result and adding them in the fixed-point domain [107, 104], losing computational density. Others try to compensate by using longer stochastic stream lengths at the cost of throughput and energy efficiency [204]. ACOUSTIC showed OR-accumulation using split-unipolar stochastic streams to be a viable, unscaled accumulation approach for neural network acceleration, but it still suffers from accuracy loss.

Stochastic bitstream generation has also received much attention. A typical stochastic number generator (SNG) has a random number generator (RNG) and a comparator that compares the target value with the random number [100]. Streams generated from a true random number generator (TRNG) have a predictable error variance that can only be reduced through longer stream lengths [102]. Less expensive TRNGs [205, 206, 207, 208] as well as quasi-random sequences [107, 104, 110] have been explored to reduce error and cost of stream generation. Correlation of the random sources in stream generation can severely impact accuracy and has forced most prior works to limit the amount of computation performed in the stochastic domain, sacrificing potential performance benefits [107, 104, 110]. In this work, we show that those sacrifices are not necessary.

Further, most prior SC literature focuses on SC "component" improvements [107, 104, 64] or implement dedicated network-specific accelerators [106, 105]. Programmable, full-system SC implementations [166, 5], like the focus of our work are rare. We account for

overheads of generalizability of programmable accelerators and generate power, performance, and accuracy numbers for the entire compute+memory system.

In this Chapter, we propose GEO - Generation and Execution Optimized stochastic computing for neural networks - an ensemble of optimization techniques that can bridge the accuracy gap between stochastic and fixed-point accelerators while improving inference energy and latency even when compared to the state of the art stochastic inference accelerators. Our contributions are as follows:

- We show that, with appropriate training, neural networks can learn the biases caused by pseudo-RNGs and extensive sharing of them in SNGs and *improve* accuracy compared to using non-shared TRNGs by as much as 6.1% points while reducing energy and area.
- We propose a progressive stream generation and shadow buffering scheme that reduces required memory bandwidth by up to 4X while improving latency by as much as 2X.
- We propose using a balanced mix of stochastic OR and fixed-point accumulation to improve accuracy by up to 9.4% points. The increase in accuracy allows us to reduce stream length by 4X while maintaining 2.2-4.0% points accuracy advantage.
- We leverage pipelining and near-memory computation to enable high throughput, maximal reuse, and efficient compute utilization regardless of layer parameters.
- Overall, we show that GEO is Pareto-superior to existing SC-based and fixed point accelerators in accuracy and energy/latency tradeoffs.

The rest of this Chapter is organized as follows. Section 4.2 describes the proposed GEO improvements to stream generation and Section 4.3 describes the optimizations to computation and architecture. Experimental methodology and results are discussed in Section 4.4 and we conclude in Section 4.5.

4.2 Stochastic Stream Generation Optimizations

This section proposes a methods optimizing the stream generation process of SC. Combining shared stream generation and training improves accuracy, while progressive generation relieves memory bottleneck.

4.2.1 Co-optimized Shared Generation and Training

RNG Sharing has been shown to be detrimental to stochastic computing accuracy[175, 209], and typically requires complicated methods to decorrelate streams from the same source to avoid incurring large stream generation penalties. However, we hypothesize that a partially-shared generation leads to higher accuracy, especially when coupled with deterministic stream generation and stream-based training.

Deterministic and repeatable (using a pseudorandom RNG) stream generators guarantee obtaining the same outputs from the same inputs, enabling the model to train for a fixed instead of a random error. We achieve determinism using maximum-length linear feedback shift registers (LFSR) as RNGs. When generating streams of length 2^n, an n-bit maximum-length LFSR is used with a cycle of $2^n - 1$. Apart from guaranteeing an almost accurate generation, LFSR generates the same output with the same input and seed and allows multiple uncorrelated stream generations (by varying the seed or the characteristic polynomial) suitable for large multiply-accumulate operations. Sharing stream generation simplifies the error profile caused by SC. Assuming that all kernels in a layer share the same set of seeds, training only needs to deal with an error associated with one set of seeds.

To test this hypothesis, we implement three levels of sharing for a 4-layer CNN[95] on the SVHN dataset. Streams are represented using the split-unipolar format, and OR is used for accumulation, similar to ACOUSTIC. In the "no sharing" case, each SNG gets a different seed for its LFSR. The "moderate sharing" case shares the same set of seeds across all kernels in a given layer. Finally, in the "extreme sharing" case, all rows of all kernels in a layer use

the same set of seeds. The same is done when a true random number generator (TRNG) is used as an RNG[1]. The results are shown in Figure 4.1. *At moderate sharing levels, LFSR-based SNGs show a significant uplift in the accuracy (up to 6.1% points compared to unshared TRNGs) at both stream lengths*, adhering to the hypothesis. TRNG does not see the accuracy improvement with sharing due to the lack of determinism. However, both TRNG and LFSR suffer from a significant drop in accuracy when using extreme sharing. In this case, stream correlation becomes an issue hard to overcome just by training.

These results also mean that low discrepancy (LD) sequences are not suitable for OR accumulation due to the difficulty of generating multiple uncorrelated streams, even though LD sequences can improve accuracy for single operations [210]. We also compared the validation accuracy when using LFSR without modeling it during training. The models are trained using TRNG but validated using LFSR. No accuracy can be gained from moderate sharing when the model is not trained for it, and extreme sharing reduces accuracy to about 20%. We use the moderate sharing scheme in GEO (up to the limit of availability of unique RNG seeds).

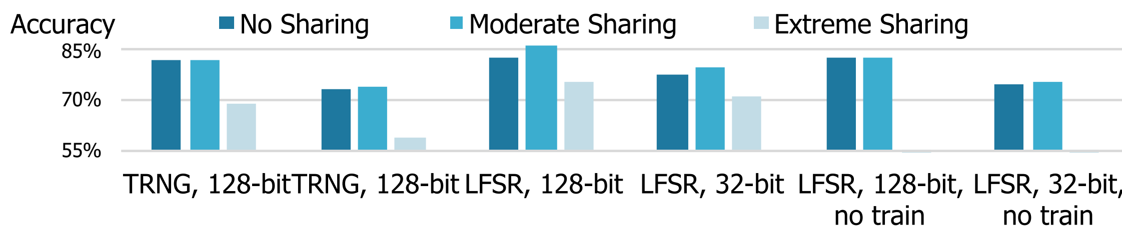


Figure 4.1: Accuracy vs. sharing for TRNG and LFSR-based random number generation.

4.2.2 Progressive Stochastic Stream Generation

Once stream computation is finished for a given set of weights and activations, the SNG buffers need to be reloaded for the next iteration. If the underlying architecture needs

[1]Due to the lack of hardware TRNG, we approximate it using the rand function in PyTorch.

to reload activations and weights extensively during computation, reloading can become a significant bottleneck. We propose using a progressive generation scheme to alleviate this inefficiency, where stream generation begins as soon as the first two most-significant bits are loaded into the buffers instead of waiting for all 8 bits, as shown in Figure 4.2. The rest of the bits are padded with 0s. As stream generation continues, the remaining bits are gradually loaded in groups of 2 bits every two cycles until the number of bits loaded matches the LFSR length used. As GEO matches the LFSR length to the stream length being used, shorter stream lengths effectively truncate the converted fixed-point values. Our progressive buffering scheme can take advantage of that truncation to reduce the number of required memory accesses, which is not possible when all bits for a given value are being loaded in parallel. Compared to starting generation when all 8 bits are loaded, *progressive generation reduces the latency overhead of reloading by 4X.*

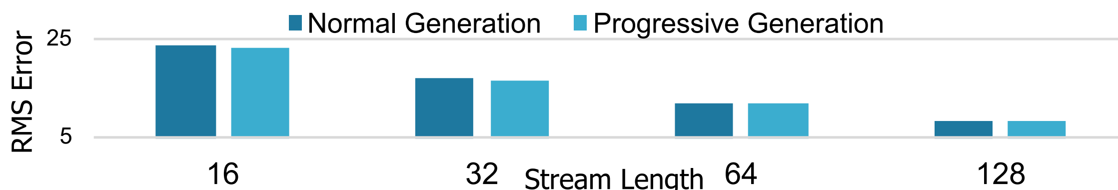


Figure 4.2: Accuracy comparison between normal generation and progressive generation performing a multiplication of two uniformly sampled inputs. RMS Error is multiplication error compared to an 8-bit integer.

As shown in Figure 4.3, performing progressive loading does not hurt multiplication accuracy. Generation is accurate after eight cycles at most when the loaded values match LFSR length. Progressive loading introduces error in at most eight cycles when using 7-bit lfsr and 128-bit streams. On a network level, using progressive loading only lowers accuracy by 0.42% when using 32-bit streams and 0.16% when using 64-bit streams. Note that this is a worst-case scenario where all input and weight streams are assumed to be generated progressively. Any input or weight reuse in the architecture leads to fewer reloads.

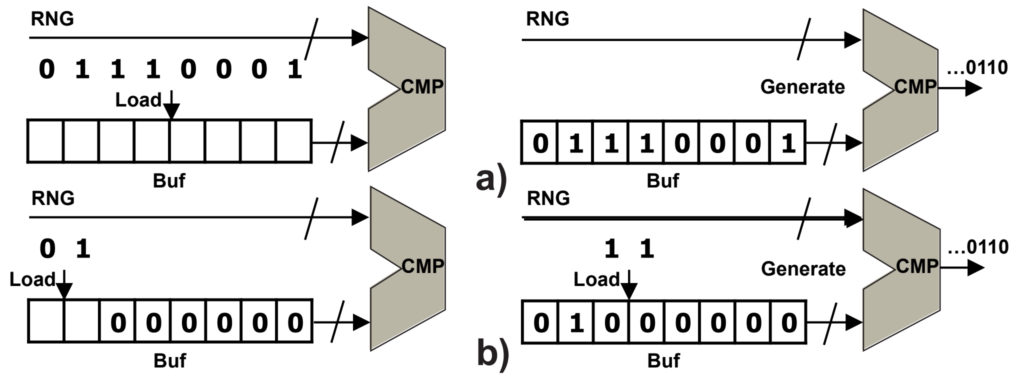


Figure 4.3: Normal SNG (a) and progressive stream generation (b).

4.3 Stochastic Computing Execution Optimizations

This section describes the overall GEO architecture and discusses a variety of microarchitectural optimizations to improve performance and accuracy on GEO.

4.3.1 GEO Architecture

Before describing further execution optimizations, we will briefly explain the underlying accelerator architecture. The GEO accelerator uses fully-stochastic computation, which can easily be modified to support different levels of partial-binary accumulation. Further, it is agnostic to the stream generation scheme and supports extensive RNG sharing. We will now briefly describe the architecture functionality. GEO architecture is largely similar to ACOUSTIC, except for the introduced optimizations.

Figure 4.4a) shows the block diagram of the accelerator. It uses separate *weight and activation memories*, which are used to load their respective *SNG buffers*. Both weight and activation memories are organized in 2 logical banks, supporting ping-pong operation. For weights, this allows loading the next set of kernels from external memory, while the current one is being processed. For activations, it enables loading activations while writing back partial sums and outputs. Both sets of memories are sized accordingly to support such

operation.

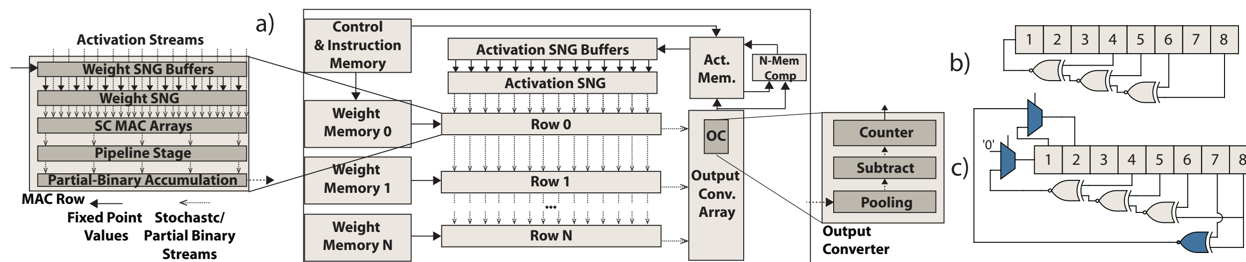


Figure 4.4: Overall SC accelerator architecture block diagram. with breakdowns of the MAC row (left) and output converter (right) modules (a). Fixed 8-bit maximum length LFSR (b), and configurable 8- or 7-bit maximum length LFSR (c).

Once all required inputs and weights are loaded into the buffers, the stream generation begins, and SNG outputs are fed directly into the compute engine. The compute is organized to maximize density while minimizing the conversion costs of stochastic streams. It is logically partitioned into *rows*, where each row is responsible for one output channel. This way, the same set of activations can be broadcasted across multiple rows, amortizing activation stream generation costs. Within each row, the same set of weights is multiplied with different sets of activations, emulating the convolutional sliding window. This way, the architecture also achieves high levels of weight reuse.

Output streams of each row are passed to the *output converter array*, where individual *output converter* modules convert them to a fixed-point format to accumulate the final value into a counter. By using small, configurable parallel counters before the conversion, the output converter array can add neighboring outputs, achieving average pooling with computation skipping on layers followed by pooling operators as in [5]. Computation skipping allows shorter streams on layers with pooling since average pooling adds multiple streams in the fixed-point domain. Once the stream generation is finished and output values are completely accumulated, they are passed through the near-memory batch normalization and

ReLU activation blocks before being written back to activation memory to serve as inputs to the next layer.

4.3.2 Partial Binary Accumulation

As mentioned in Section 4.1, many recent SC works opt to perform accumulation in the fixed-point domain, as it offers higher accuracy than SC-based addition [106, 107]. In contrast, few others have tried implementing fully-stochastic accumulation to save costs. In contrast to these two extremes, we propose to use partial SC-fixed-point accumulation, where the first few levels of accumulation are implemented in SC using OR gates before converting the intermediate results to fixed-point and adding them.

The partition between SC and fixed-point accumulation significantly affects both accuracy and performance. While using an approximate parallel counter (APC) [211] allows one layer of SC accumulation before fixed-point accumulation, the combined use of AND and OR makes it equivalent to multiplexers and is thus unsuitable for multiple layers of accumulation. Using OR for addition with training allows an arbitrary trade-off between SC and fixed-point domains. We tested model accuracy with different fixed-point accumulation levels. Assuming weight filters are arranged into (C_{in}, H, W) dimensions, *performing fixed-point accumulation in the W dimension (PBW) improves accuracy by 4.5% and 9.4% respectively for 128-bit and 32-bit streams compared to performing all accumulations using OR*. Extending fixed-point accumulation to H (PBHW) improves accuracy by $< 0.5\%$ but increases the number of fixed-point adders by 5X for 5×5 filters.

Adding support for partial binary accumulation only requires replacing the last levels of OR accumulation with a parallel counter. While the level of partial binary accumulation is fixed at the design stage, it still allows for trading off precision with latency through SC stream length configuration. Since partial binary accumulation fabric operates on a bitwise basis, it is agnostic to the chosen stream length. Parallel counters in the average pooling fabric in the output converters need to be adjusted to handle wider inputs. In Section 4.4,

we show that those changes have minimal impact on the overall architecture.

Figure 4.5 shows the overhead, in terms of area, of implementing SC-based MAC units with a partial binary accumulation stages. We compare the full-or accumulation (SC), PBW, PBHW, and fixed-point accumulation (FXP) configurations for different three-dimensional kernel sizes. While area overhead of PBW and PBHW partial binary accumulation can be as much as 1.4X and 4.5X for smaller kernels, the area increase goes down to 4% and 9% for large ones. Implementing partial binary accumulation is, therefore, well suited for highly-parallel SC architectures where such overheads would be negligible. Figure 4.5 also shows that implementing complete binary accumulation can increase the area by more than five times for most kernel sizes, emphasizing its performance limitation. While approximate parallel counters [212] (APC) offers noticeable area benefits compared to fixed-point accumulators, it is still more than 3X larger than PBW and PBHW for larger kernels. Given that PBW is almost identical accuracy-wise, the rest of the paper uses PBW as the default unless otherwise mentioned.

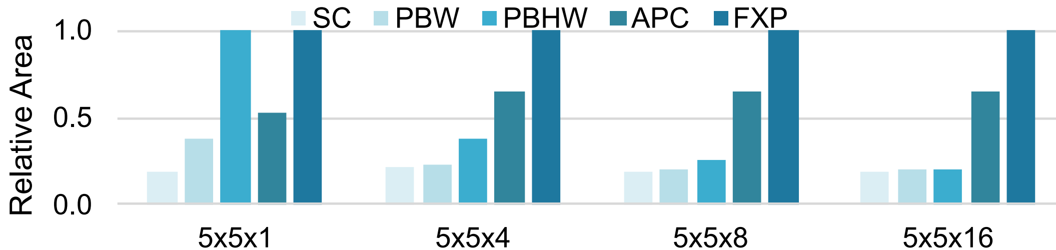


Figure 4.5: Area comparison for different hardware implementations of SC-based MAC units for different kernel sizes and different levels of partial binary accumulation.

Using partial binary accumulation increases the dynamic range of outputs. Since the increase of output precision comes primarily from increased range, truncating activations without factoring in the dynamic range diminishes partial binary accumulation benefits. To deal with this, we use an 8-bit fixed-point version of batch normalization (BN) before ReLU activation to minimize the cost of implementing it in hardware. While still potentially expensive, BN offers 5.5-6.5% points accuracy improvement. For layers with pooling, pooling

is placed before ReLU activations so that BN can be performed on pooled activations.

4.3.3 Near-Memory Computation

Organizing the GEO accelerator compute hierarchy to mimic a vertically sliding convolutional window means that it naturally yields to the weight-stationary dataflow [2]. While the window iterates through the output tensor, weights can stay unchanged, and only a single row of activations needs to be reloaded between each computational pass, therefore minimizing both weight and activation bandwidth requirements. Indeed, this dataflow choice reduces the overall number of memory accesses by up to 3.3X compared to input-stationary, making it the optimal choice in virtually every convolutional layer we have explored. However, this is only true if a strict, weight-stationary implementation can be enforced. It requires that the MAC units' width and the corresponding number of SNGs are sized to fit the entire activation tensor covered by a kernel in a given layer. This constraint guarantees that output values can be generated during a single computational pass, without partial sums, effectively "merging" weight- and output-stationary dataflows in one.

However, it is not uncommon in modern neural networks to find kernels with thousands of weights, which cannot be fully unrolled without sacrificing a prohibitive amount of silicon area. When that is not possible, the accelerator needs to store converted partial sums for later accumulation. If the architecture does not support that, it has to implement a strict output-stationary dataflow, accumulating output values in output conversion modules over multiple passes, where both weights and activations need to be swapped between each pass. Such dataflow can increase memory accesses by as much as 10.3X vs. ideal, weight-stationary implementation. While progressive generation can alleviate such dataflow's bandwidth requirements to a degree, the steep energy cost of memory accesses remains.

One way to avoid being forced into such suboptimal dataflow is to couple output conversion modules with small register files. However, the number of registers required will depend on a particular layer - to support some of the very deep ones would require register files that

dwarf the size of conversion modules. At the same time, those register files would remain mostly unused on the shallower layers. Instead, *we propose implementing near-memory accumulation, where the activation memory is tightly coupled with an array of adders.* We then expand the GEO ISA to support a 2-cycle read-add-write vector instruction that can be used to accumulate partial sums. Since partial sums are stored in large activation memory, there is no need to size it for any specific network or layer.

There are two downsides w.r.t. to local register files. First, activation memory accesses are much more energy costly than to local registers. However, in this dataflow, partial sum accesses constitute only 13% to 20% of overall memory accesses, meaning they are not critical to overall energy consumption. Second, additional accesses put more strain on memory bandwidth. However, as we will show in Section 4.4, progressive shadow buffering can alleviate this problem. We further expand this scheme to support near-memory batch normalization through an array of fixed-point MAC units, tightly coupled with activation memory.

4.3.4 Pipeline Optimizations

On top of the generation and execution optimizations listed above, the GEO accelerator includes two microarchitectural enhancements. First, we supplement the progressive generation with shadow buffers. When current progressive values are fully loaded, a certain number of bits can be loaded into the shadow buffers for the next computation. Thanks to that, the following computation phase can begin immediately after the current one finishes, since the minimum number of bits required, which in our case is two, is already available in the shadow buffer. Without progressive generation, shadow buffers would need to be the same size as the actual SNG buffers (i.e., 4X larger), incurring a significant area penalty. The overhead of progressive shadow buffers is only about 4% at the whole accelerator level.

Second, we implement a pipeline stage within our compute engine between the SC and partial-binary accumulation stages. This is because of a long critical path between the LFSR,

SNG, SC MAC, partial binary accumulation, and output counters. *Implementing the pipeline stage in that location allows us to cut down the critical path by over 30% while minimizing the area required by additional flip-flops* (<1% overhead on the accelerator level). Because of the recovered timing slack, we can now reduce the operating voltage without lowering the frequency to achieve better energy efficiency.

4.4 Evaluation & Results

4.4.1 Evaluation Methodology

We test accuracies on CIFAR-10, SVHN, and MNIST datasets. For CIFAR-10 and SVHN, we use the same 4-layer CNN [95] (CNN-4) as in Section 4.3 and VGG-16 [1]. VGG-16 has the X/Y input dimensions of each layer downscaled, and the fully-connected layers are reduced to FC-512 instead of FC-4096 to accommodate the smaller image sizes. For MNIST we use LeNet-5 [152]. We use PyTorch 1.5.0 to train the models. We implement the forward pass using both floating-point and simulated SC. Simulated SC is used to compute output values, while the floating-point forward pass is used to guide backpropagation. With SC simulation's speed limitations, we skipped training for more complex datasets (i.e., ImageNet) due to the prohibitively long training time. Due to the use of floating-point for backpropagation, GEO can only accelerate the inference of SC models. Models are trained using the ADAM optimizer [153] with an initial learning rate of 2e-3, and accuracy is evaluated on the corresponding testing dataset after 1000 epochs. Each model is trained with different stream lengths using split-unipolar implementation and designated by two stream lengths $\{s_p - s\}$, s_p for layers with pooling and s for layers without it. While max pooling is possible, we use average pooling with computation skipping to reduce stream length requirements for layers with pooling. Output layers always use 128-bit streams due to their small performance impact but noticeable accuracy benefits. The actual stream length used is double the specified value due to the use of split-unipolar representation.

To estimate the area, power, and latency of the proposed design, we have written individual blocks (SNGs, MAC arrays, buffers, output converters) in Verilog and then synthesized them using a commercial 28nm library. Memories were modeled using CACTI 6.5 [177]. For the LP variant described below, we consider the cost of external memory accesses, with the bandwidth and access energy modeled after the HBM2 standard [213]. We used activity factors obtained through RTL simulations to adjust active power numbers in synthesis (since many modules, such as SNG buffers and batch normalization modules are idle most of the time). To obtain accurate energy and latency estimates, we used a custom performance simulator, which combines the numbers from individual modules with a compiled code representing the given network model. Since the proposed enhancements are mostly agnostic of the control flow, we use the ISA used by ACOUSTIC, with minor modifications. We create two versions of GEO: ultra-low power (ULP) or low-power (LP) targeted at different area points and network sizes. ULP has 25.6K MACs with total on-chip memory of 150KB, while the LP variant has 294K MACs and 0.5MB of on-chip memory.

As a fixed-point baseline, we use Eyeriss [13], scaled to 4-bit or 8-bit precision and 28nm node. The on-chip memory capacity and the number of processing elements are chosen to achieve close to the iso-area comparison point with GEO. We simulate the execution of the neural networks using [178]. For SC comparison points, we use the ACOUSTIC [5], Sign-Magnitude SC (SM-SC) [109] and SCOPE [166]. ACOUSTIC configurations are sized to have the same amount of memory and compute as GEO, and we use longer stream lengths to maintain close to iso-accuracy with GEO. ACOUSTIC architecture configurations differ from the original, but we use the same simulation framework, ensuring consistent results. We also include activity factors in power estimation. SM-SC is not a fully programmable accelerator making full comparison impossible. SCOPE is an in-memory, DRAM-based accelerator with a massive area footprint, not well suited towards edge applications [166]. Unfortunately, many recent works on SC neural network acceleration only report performance numbers for the compute part while omitting the crucial impact of memory and dataflow, making it

impossible for us to compare on the system level. Numbers are scaled to the 28nm node when necessary, using the models provided in [214]. We further compare GEO-ULP with CONV-RAM [7] and MDL-CNN [6] mixed-signal accelerators.

4.4.2 GEO Accuracy Comparisons

Table 4.1 compares the accuracy of GEO with fixed-point and other SC implementations. Eyeriss results are retrained at respective precision[2]. Results for other works are reported from the respective papers. *GEO offers 2.2-4.0% points better accuracy at quarter stream length compared to ACOUSTIC and similar accuracy at the same stream length compared to [109].* Both shared stream generation and partial binary accumulation contribute to increased accuracy. For CNN-4 on SVHN with 32-64 stream length, dropping binary accumulation lowers accuracy to 79.6%, while using TRNG on top of that drops it further to 73.7%. Compared to fixed-point, the accuracy with CNN-4 is comparable to 4-bit fixed-point when using 32-64 setup on SVHN, but 4% lower on CIFAR-10 when using 32-64 and 1.9% lower when using 64-128[3]. Accuracy with VGG-16 is 2.2% lower than the 8-bit fixed-point on CIFAR-10 and comparable on SVHN. Accuracy on MNIST is already comparable to fixed-point in the baseline, and GEO optimizations do not affect it. Compared to CONV-RAM[7], an in-memory architecture and MDL-CNN[6], a time-domain architecture, GEO offers superior accuracy even with 16-32 stream length.

4.4.3 Performance Impact of GEO Enhancements

We compare the baseline ULP architecture (without GEO optimizations and 16-bit LFSRs to emulate TRNG) with two GEO variants:

---

[2]Original Eyeriss [13] was 16-bit with truncated accumulation, which suffers from substantial accuracy loss at lower 4/8-bit precision. We assume full 16-bit accumulation bitwidth and, as a result, Eyeriss accuracy results are somewhat optimistic.

[3]While intermediate accumulation results for Eyeriss are allowed to have double the input precision, overflow may still happen and these results are optimistic

Table 4.1: Accuracy comparison with fixed-point, other SC implementations and so on.

		Eyeriss		ACOUSTIC		GEO			SCOPE[166]	CONV-RAM[7]	MDL-CNN[6]	SM-SC[109]
Dataset	Model	8-bit	4-bit	256	128	64-128	32-64	16-32	128	7a1w	4a1w	128
CIFAR-10	CNN-4	85.1%	82.1%	78.0%	74.9%	**80.2%**	**78.1%**	—	—	—	—	80%
	VGG-16	90.9%	—	—	—	**88.7%**	**88.7%**	—	—	—	—	—
SVHN	CNN-4	93.3%	90.5%	89.0%	86.8%	**91.9%**	**90.8%**	—	—	—	—	—
	VGG-16	96.2%	—	—	—	**96.0%**	**95.9%**	—	—	—	—	—
MNIST	LeNet-5	—	99.3%	—	99.3%	—	**99.3%**	**98.9%**	99.3%	96%	98.4%	—

- GEO-GEN-128,128 - uses the generation optimizations from Section 4.2. Progressive shadow buffers are used in this configuration.
- GEO-GEN-EXEC-32,64 - uses both the generation and execution (from Section 4.3 optimizations. Further, it reduces the stream lengths being used to remain iso-accuracy with other configurations.

The area, energy, and latency impacts of GEO optimizations on the ULP architecture are shown in Figure 4.6. For energy and latency, we simulated the SVHN CNN inference on each of those design points. Generation optimizations result in an overall 1% decrease in the accelerator area, where an increase in the area due to progressive shadow buffers is balanced by more extensive RNG sharing. At the same time, *the use of progressive shadow buffers to hide memory latency results in a 1.7X speedup and 1.6X reduction in energy.* Energy savings come mainly from SNG optimizations and reduced leakage.

Adding execution optimizations on top of the generation ones increases the area by 2% w.r.t. to baseline. The impact of pipelining and partial binary accumulation is minimal due to its limited application and an overall small contribution of the SC MAC array to the overall area. Similarly, near-memory computation is well amortized because it is time multiplexed. *The combination of shorter stream lengths, more efficient dataflow enabled by near-memory computation and pipelining coupled with DVFS results in 4.3X and 5.2X reduction in latency and energy w.r.t. baseline.*

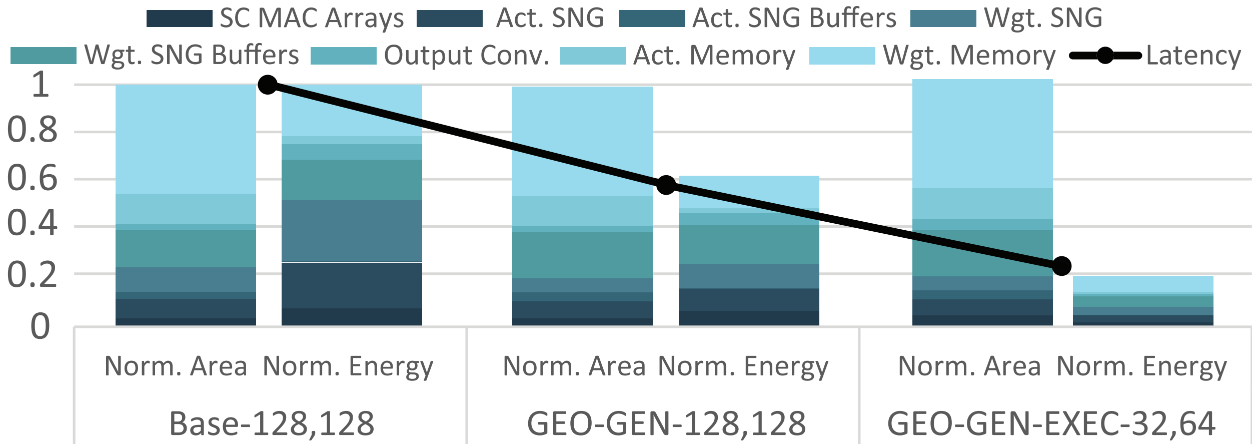


Figure 4.6: Area, energy and latency for different GEO configurations (normalized to Base-128,128).

4.4.4 GEO Performance Compared

Table 4.2 shows a comparison of the proposed GEO ULP accelerator with fixed-point and mixed-signal approaches. First, we show that GEO-32,64 outperforms the 4-bit fixed-point baseline in terms of throughput, by 2.7X, and energy efficiency, by 2.6X, in the same area. It also outperforms ACOUSTIC-128, by 4.4X and 5.3X, respectively, while achieving higher accuracy. It is also highly-competitive in terms of energy-efficiency with mixed-signal accelerators like Conv-RAM and MDL-CNN. We refrain from comparing the throughput against those implementations due to the large area difference.

On the scale-out end of the spectrum, GEO LP outperforms iso-area, 8-bit Eyeriss by 5.6X in terms of throughput and 2.6X in terms of energy efficiency. Modest energy reduction is caused by the high cost of external memory accesses - when those are omitted, GEO is as much as 6.1X more energy-efficient than Eyeriss. It is also 2.4X faster and 1.6X more energy efficient than ACOUSTIC while having higher inference accuracy. Despite occupying only 3.3% of SCOPE area, GEO has nearly 24% of its peak throughput and has 2.4X better energy efficiency.

Table 4.2: Comparison between GEO ULP and fixed-point and neuromorphic implementations. Numbers are scaled to 28nm.

	Eyeriss 4-bit [2]	**GEO ULP -32,64**	Conv-RAM [7]	MDL CNN [6]	ACOUSTIC ULP-128 [5]	**GEO ULP -16,32**
Voltage	0.9	**0.81**	0.9	0.537	0.9	**0.81**
Area [mm^2]	0.59	**0.58**	0.02	0.06	0.57	**0.58**
Power [mW]	20	**48**	0.016	0.02	72	**48**
Clock [MHz]	400	**400**	364	25	400	**400**
Precision	4-bit	—	6b/1b	8b/1b	—	—
CIFAR-10 Fr/s	5.2k	**14k**	—	—	3.2k	**29k**
CIFAR-10 Fr/J	115k	**305k**	—	—	57k	**576k**
LeNet5 CNN Fr/s	47k	**520k**	15k	1k	3.2k	**780k**
LeNet5 CNN Fr/J	790k	**42M**	40M	33.6M	57k	**56M**
Peak GOPS[4]	80	**640**	10.7	0.365	160	**1280**
Peak TOPS/W	4	**13.3**	44.2	18.2	2.22	**26.6**

4.5 Conclusion

In this Chapter we presented GEO, a generation and computation-optimized stochastic computing architecture for neural network acceleration. We develop an ensemble of accuracy improvement and energy/runtime improvement techniques. These optimizations improve accuracy by 2.2-4.0% points compared to state-of-the-art SC-based accelerators while also being 4.4X faster and 5.6X more energy efficient. GEO can compete with fixed-point implementations with similar accuracy and area while delivering up to 5.6X throughput and 2.6X energy-efficiency gains. GEO, despite being an all-digital, programmable accelerator, can achieve energy efficiency comparable to in-memory/mixed-signal accelerators.

Table 4.3: Comparison between GEO LP and fixed-point and SC implementations. Numbers are scaled to 28nm.

	Eyeriss 8-bit [2]	**GEO LP -64,128**	SM-SC [109]	SCOPE [166]	ACOUSTIC LP-256 [5]	**GEO LP -32,64**
Voltage	0.9	**0.81**	0.9	—	0.9	**0.81**
Area [mm^2]	9.3	**9.2**	—	273	9	**9.2**
Power [mW]	848	**797**	—	—	1160	**797**
Clock [MHz]	400	**400**	1536	200	400	**400**
CIFAR-10 VGG Fr/s	555	**3.1k**	—	—	1.3k	**5.2k**
CIFAR-10 VGG Fr/J	618	**1.6k**	—	—	1k	**2.2k**
Peak GOPS	204	**1.8k**	1.7k	7.1k	460	**3.6k**
Peak TOPS/W	0.48	**2.25**	0.92	—	0.4	**4.5**

CHAPTER 5

SASCHA - Combining Randomness with Sparsity

ACOUSTIC, GEO, and other works have shown the potential of using stochastic computing for machine learning acceleration. Its high compute density, affinity with dense linear algebra primitives, and approximation properties have an uncanny synergy with deep neural network computational requirements. However, there is a conspicuous lack of work trying to integrate SC hardware with sparsity awareness, that has brought significant performance improvements to conventional architectures. In this Chapter, we identify why popular sparsity-exploiting techniques are not easily applicable to SC accelerators and propose a new architecture - SASCHA - Sparsity-Aware Stochastic Computing Hardware Architecture for neural network acceleration that addresses those issues. SASCHA encompasses a set of techniques that make utilizing sparsity in inference practical for different types of SC computation. At 90% weight sparsity, SASCHA can be up to 6.5X faster and 5.5X more energy-efficient than comparable dense SC accelerators with a similar area without sacrificing the dense network throughput. SASCHA also outperforms sparse fixed-point accelerators by up to 4X in terms of latency. To the best of our knowledge, SASCHA is the first stochastic computing accelerator architecture oriented around sparsity.

Collaborators:

- Tianmu Li, Electrical and Computer Engineering, UCLA.
- Professor Puneet Gupta, Electrical and Computer Engineering, UCLA.

5.1 Introduction

New classes of machine learning mobile applications, like virtual assistants, translation, and image recognition, continue to emerge, amplifying the demand for fast, efficient, and secure inference [32, 33, 34]. Increasingly, this demand cannot be satisfied by offline, cloud-based processing due to substantial and unpredictable network latencies, as well as privacy concerns [32, 215]. To enable online machine learning, mobile devices increasingly incorporate custom accelerators, broadly known as neural processing units, or NPUs [34]. Those devices are deployed under strict area, power, and energy constraints.

To improve the throughput and energy efficiency, researchers have increasingly looked into model compression methods, like quantization and pruning [35, 20, 21, 90], including 3PXNet presented in Chapter 2. At the same time, non-conventional computing methods, like in-memory or stochastic computing, have also been gaining popularity [102, 197, 133]. Hardware support for some of those techniques has already made its way into commercially available devices [216].

As demonstrated in Chapters 3 and 4, stochastic computing, in particular, is a promising approach to approximate computing acceleration, particularly for dense, compute-heavy models like convolutional neural networks [10, 105, 64]. There is now a plethora of possible SC *flavors*, spanning the accuracy-efficiency Pareto curve, depending on application requirements. However, there is a conspicuous lack of SC architectures trying to take advantage of neural networks' resilience to pruning [21], something that could enable further performance improvements. To address this gap, we propose SASCHA - Sparsity-Aware Stochastic Computing Hardware Architecture for Neural Network Acceleration. SASCHA consists of computational unit design, accelerator architecture, and a scheduling method that improves the efficiency of executing sparse neural networks using SC computation without sacrificing high parallelism and data re-use opportunities.

The key contributions of this work are:

- We introduce the multi-group, parallel stream sparse SC SASCHA processing element (PE), agnostic of underlying SC computation style, and perform a thorough evaluation of its extensive design space.

- To the best of our knowledge, we propose the first stochastic computing neural network accelerator architecture that takes advantage of parameter sparsity. SASCHA can achieve up to 6.5X throughput and 5.5X energy efficiency improvement at 90% sparsity level, compared to a dense SC accelerator with a similar area, while maintaining the throughput and suffering only up to 31% energy efficiency in the dense case.

- We propose a weight bit-slicing technique using asymmetric streams unique to SC that can extract weight sparsity even in dense networks, improving SASCHA throughput and energy efficiency by up to 1.75X on unpruned networks.

The rest of this Chapter is organized as follows: Section 5.2 explains why conventional approaches to exploiting sparsity are not easily applicable or beneficial in the case of SC accelerators. Section 5.3 introduces the sparse SASCHA PE and explores its design space. Section 5.4 describes the architecture of the SASCHA accelerator and its asynchronous scheduler. Section 5.5 discusses a bit-slicing method that can extract high effective sparsity in unpruned networks. Section 5.6 shows the benefits of the SASCHA accelerator. Finally, Section 5.7 summarizes related work.

5.2 Motivation

Given the recent popularity of stochastic computing and sparsity-aware accelerators, there is a surprising lack of attempts to combine both approaches. This section explains why taking advantage of sparsity in stochastic computing hardware cannot be tackled by the same techniques as conventional, floating- or fixed-point accelerators.

Most common attempts to exploit sparsity in hardware rely on matching non-zero input

and weight values that need to be multiplied together to avoid ineffectual computations, i.e., ones where at least one operand is zero [217, 218, 17, 219, 79, 126, 90, 220, 127]. To avoid an issue where no non-zero operands are available, causing stalls and poor utilization, larger staging buffers that can spatially or temporarily *advance* operands are frequently used [217, 218, 17, 219]. While pre-trained weights can often be scheduled offline, simplifying the hardware, exploiting input sparsity must be performed dynamically, incurring non-negligible hardware overheads. For example, [219] increases the area of the compute core by 2.8-5.9x to support sparsity compared to the dense baseline. In other approaches, matching is achieved by calculating an intersection operation between input and weight values on an output-by-output basis [79, 127, 220], or coupling custom dataflows with sparse storage format choices [221, 222]. Those techniques can also incur significant overheads. For example, in [127], the intersection calculation module is more than 10x larger than the compute. In [79], a large crossbar network is required to route the outputs, exceeding the size of compute units by more than 3X. Similarly, in [222], more than 95% of the processing element area is consumed by the sorting queues.

For devices operating with conventional floating- or fixed-point values, high area and energy cost of computation can justify such overheads. However, that is not the case in SC-based architectures. As mentioned previously, conversion circuitry, including SNGs, RNGs, and buffers, is frequently the dominant area and power contributor in SC accelerators. Analyzing the accelerator area breakdowns in [10], we can see that the SC MAC arrays (multipliers and adders) occupy as little as 6% of the overall area, while the conversion circuitry consumes 51%, with similar energy contributions. While the techniques mentioned above could be applied to SNGs, the high spatial data reuse introduces an additional level of complexity [5, 104, 105]. Commonly, thousands (or more) of multiply-accumulate operations can be scheduled concurrently, with individual operand streams being broadcast across many multiplications in parallel. Therefore, a given operand could only be skipped if all corresponding operands it is meant to be concurrently multiplied with are zeros. Such ex-

tensive reuse would limit ineffectual computation skipping opportunities and dramatically increase the cost of already expensive dynamic intersection calculation, dwarfing low-cost SC computation. Further, conventional sparse accelerators require complex sparsity detection logic to improve the utilization of the limited number of their large and complex processing elements [219]. SC's high parallelism and low cost make underutilization less of a problem compared to architectures with a limited number of large floating- or fixed-point PEs [5]. In short, for SC architectures, *detecting sparsity is more costly than ignoring it.*

Taking advantage of sparsity in stochastic computing architectures by using conventional approaches can, therefore, yield minimal benefits or could end up being detrimental. To this end, we make a few guiding observations. First, any attempt at exploiting sparsity should not compromise the high parallelism enabled by SC [5]. Therefore any approaches requiring fine-granularity, dynamic scheduling, or other sources of hardware overheads are highly undesirable. Particularly, individual, dynamic intersections between sparse weights and activations are not compatible with spatial data reuse employed by SC. To this end, whenever possible, we want the burden of exploiting sparsity to lie on the offline, static side, so as not to introduce unnecessary hardware overheads. Because of that, we focus only on the sparsity of weights, which are static, and known a priori, while keeping activations dense. Second, explicitly avoiding the ineffectual SC computations, as opposed to floating- or fixed-point ones has limited benefits and should not be the goal in itself. Instead, we believe sparsity should be used to improve the dominant area and energy contributors in SC-based architectures: memory and SNGs [5, 10].

5.3 SASCHA Sparse SC PE

5.3.1 Sparse PE Design Objectives

In this section, we outline the design of a sparse SC processing element, implementing a parallel dot product operation. Our PE is agnostic of the underlying style of SC computation.

To demonstrate that, we evaluate three implementations: split-unipolar AND-based multiplication with partial-binary OR-based accumulation used by GEO, modified GEO-style PE with full binary accumulation, and the uMUL multiplier with binary accumulation proposed by uGEMM [64]. We will refer to them as GEO-, GEO+- and uGEMM-style PEs, respectively. Those PEs offer progressively higher precision at an increased area/power cost, as discussed in Section 5.6. While full-binary accumulation refers to adding all individual bits of SC multiplication results, preserving their full fidelity, partial-binary refers to performing the first part of accumulation using streaming, OR-based adders for more compact area, and the rest using binary accumulation [10]. Alternative SC computational components are fundamentally compatible with SASCHA, but they are beyond the scope of this work.

Our goal is to design a sparse SC compute unit, given SC hardware peculiarities outlined in Section 5.2. First, as explained in Section 5.2, our prime target for optimization is the cost of converting the numbers between fixed-point and stochastic representations. Second, we need to choose whether we should target weight or input sparsity. As explained in Section 5.2, we want to avoid any dynamic approaches that incur high hardware overheads relative to cheap SC compute. Because of that, we avoid trying to exploit both weight, and activation sparsity due to costly intersection calculations [79, 127, 126]. Even exploiting just activation sparsity would require spatial and temporal operand advancement [217, 218] performed dynamically in hardware. Instead, we focus solely on the sparsity of network weights, which is known a priori for inference accelerators and enables static, offline scheduling. It allows us to exploit sparsity with minimal hardware changes that do not compromise SC density and high spatial reuse.

5.3.2 G:C Sparse PE

Similar to [216], we exploit structural sparsity in weights. We believe this simple approach is well suited to our requirements as a) it can be statically scheduled, b) it allows us to reduce the overhead of stochastic number generation, c) it does not exploit or make any

assumptions about input sparsity, and d) does not require high multiplexing overheads, as we will show shortly. However, we make three important distinctions between the sparse compute units of [216] and our Sparse SC PE. First, we explore other sparsity structures - in general, for every group of G parameters, we allow C non-zero ones. We will henceforth refer to G as a *group size* and C as *capacity*. Second, where [216] can only accelerate networks pruned to their exact sparsity structure, we design a scheduler capable of mapping networks with arbitrary level and structure of sparsity onto the compute fabric built using sparse PEs. Finally, we focus specifically on stochastic computing, which enables different design trade-offs compared to conventional compute.

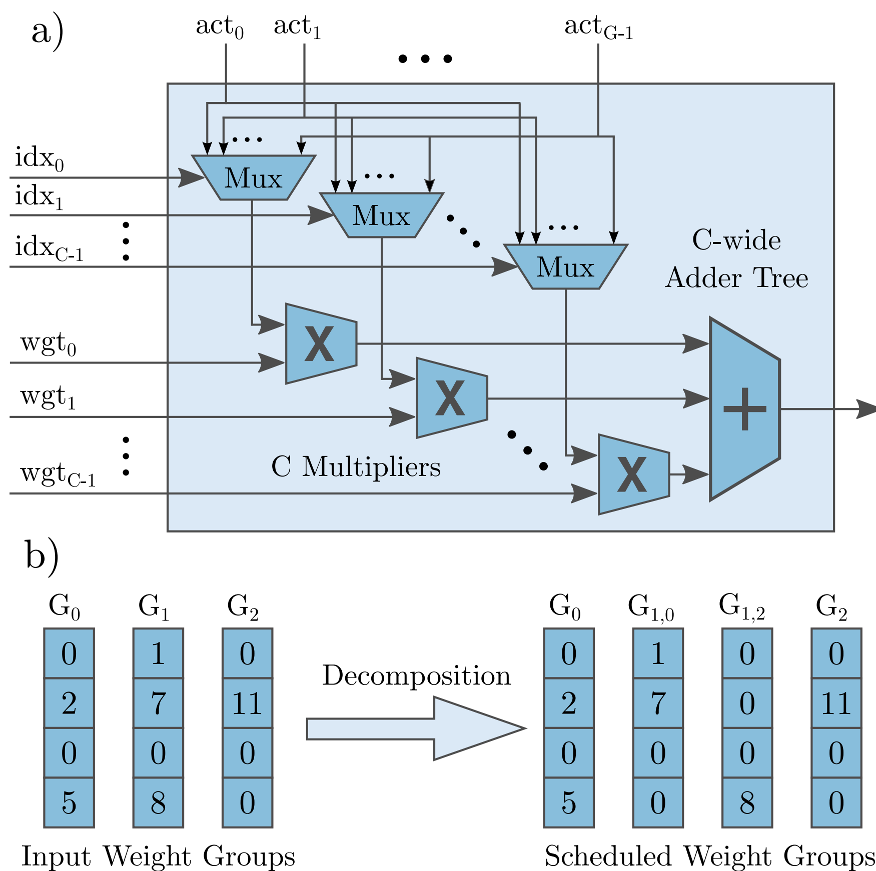


Figure 5.1: Sparse PE with group size G and capacity C (a). Decomposing 3 arbitrary parameter groups of size $G = 4$, into groups satisfying the capacity requirement of $C \leq 2$ (b).

A block diagram of a generic, i.e., supporting any format of underlying computation, sparse PE with a group size G and capacity C is shown in Figure 5.1 a). It performs a spatially parallel dot product operation between a dense vector of G activations and a sparse vector of C weights and their corresponding indices. A weight's index indicates its position in the dense vector and is used to select an activation that needs to be multiplied by the weight's value. Compared to an equivalent dense PE, it requires $G-C$ fewer multipliers and adders, at the cost of C additional $G:1$ multiplexers.

Using this PE is only possible when there is a guarantee that every group of G weights contains only up to C non-zero ones. We refer to such groups as *balanced*. When the balancing is enforced on the network parameters, as is the case with [216], with $G = 4$ and $C = 2$, the computation can be scheduled in the same way as on hardware using dense PEs while reducing storage and ineffectual operations. To schedule a network with an arbitrary level and structure of sparsity, we can decompose any weight vector of size G to between 1 and $\lceil G/C \rceil$ vectors. This is shown schematically in Figure 5.1 b), where three groups of size $G = 4$ get decomposed into four balanced groups, each satisfying $C \leq 2$. The decomposed groups can then be scheduled on a sparse PE with $G = 4$ and $C = 2$. However, this computation will require 33% longer runtime compared to a dense processing element. We refer to the ratio of the number of decomposed to original weight groups as the *iteration overhead*. The maximum iteration overhead for a given sparse PE configuration is $\lceil G/C \rceil$. It is one of the main metrics for evaluating the efficiency of different sparse PE configurations. For simplicity, we restrict both G and C to be powers of 2.

A stochastic computing equivalent using GEO-style PE [10] is shown in Figure 5.2 a). The main concern in the SC case is the overhead of additional conversion circuitry. We implemented and synthesized a set of sparse SC PEs using a commercial TSMC 28nm library and Cadence Genus synthesis tool to evaluate this. Figure 5.2 b) shows the breakdown of a fixed-point and GEO SC sparse PE with $G = 4$ and $C = 2$, excluding input and output buffers. While the fixed-point PE is dominated by the area of multipliers, in the SC one, the

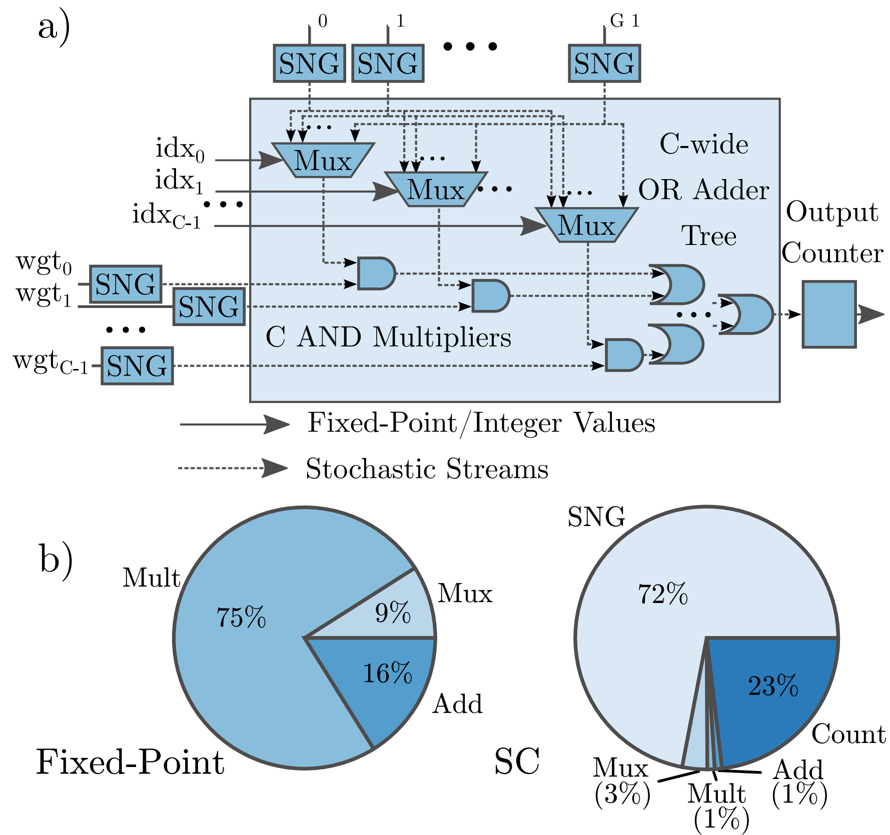


Figure 5.2: Sparse GEO-style SC PE with group size G and capacity C (a). Split-Unipolar [5] logic is omitted case for readability. Area breakdown of fixed-point (left) and GEO SC (right) sparse PEs with G=4, and C=2 (b).

arithmetic occupies only about 2% of the area. While the conversion cost can be amortized through input broadcasting and wider dot-products, it presents us with different optimization priorities compared to a fixed-point sparse PE. We have also compared the area of sparse SC PEs with different G and C, shown in Figure 5.3, for different styles of SC. For GEO-style PEs with $C < 8$, the area of the sparse SC processing element is roughly equivalent to half of the dense one, even for large group sizes. This reduction is because conversion circuitry dominates the overall area, and the sparse SC PE structure eliminates roughly half of the overall SNGs when C is small. uGEMM sparse PE shows much higher area reduction compared to dense, due to the fact that weight buffers and SNGs are bundled together with the multiplier, resulting in a larger size of a dense PE. The close coupling of stream generation and computation is done for decorrelation purposes, resulting in higher multiplication precision [64]. If the sparsity structure can be enforced, this area reduction shows great potential for synergy between SC and sparse neural networks.

5.3.3 Multi-Group Sparse SC PE

Given the proposed sparse SC PE design, we need to choose the best G and C for a sparse SC accelerator. There are three main considerations here. First, as mentioned above, is minimizing the iteration overhead. We explore it in detail in Section 5.4. The second is maximizing the hardware efficiency and amortizing SC conversion costs. SC accelerators often employ highly parallel dot product units, ranging from 16 to hundreds of concurrent MAC operations [5, 10, 64, 105]. High parallelism allows them to amortize the cost of converting streams corresponding to partial sums back to fixed-point representation. From this standpoint, using sparse PEs with large group sizes like 16 or 32 is desirable. However, this is where the third consideration comes into play - storage compression efficiency.

While we focused our discussion on sparse computation until now, another benefit of sparsity is reducing required storage - one of the main contributors to the area and energy consumption of neural network accelerators [90, 2, 15], including SC ones, [5, 10]. Sparse

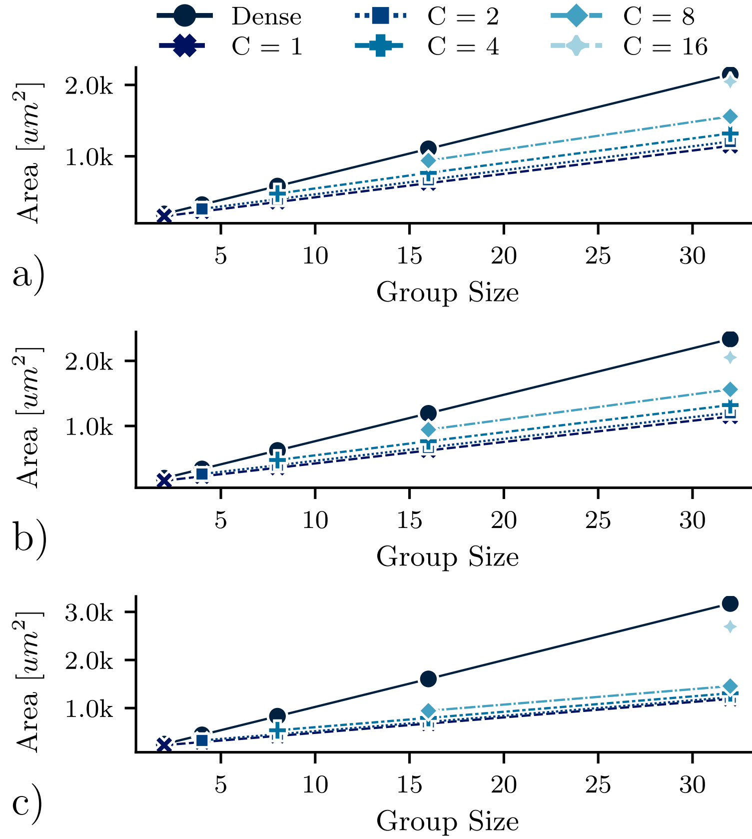


Figure 5.3: Area of sparse and dense SC PEs, given different group sizes and capacities, for GEO (a), GEO with full binary accumulation (b), and uGEMM (c) style SC.

accelerators often employ compressed memory formats like compressed sparse row (CSR) to reduce storage and throughput requirements [17, 220, 218, 222]. In this work, we explore a simple compression scheme, where each weight is coupled with an index indicating its position in the group. The group's relative position in a given filter is then handled by the scheduler as described in Section 5.4. While more efficient compression schemes may be available, they are beyond the scope of this work.

The index size is determined by the group size G and is equal to $log_2(G)$. Since indices are required on a per-value basis, they are independent of C. Larger group sizes will therefore incur higher indexing overheads. Figure 5.4 shows storage compression, the ratio between the memory required for storing all dense weights and storing only the sparse weights and

their indices. It is an ideal case, where we assume only the sparse weights are stored, and there are no additional overheads, for example, coming from alignment requirements. Using a group size of 64 requires 1.44X more storage than a group size of 2. At a sparsity level of 90%, this translates to 5.7X and 8.9X compression for $G = 64$ and $G = 2$, respectively. More importantly, when running a dense network, going from a group size of 2 to 64 reduces compression from 0.89X to 0.57X. We want SASCHA to be flexible and support networks with different sparsity levels with high efficiency, even in the dense case. Because of that, large group sizes are highly undesirable.

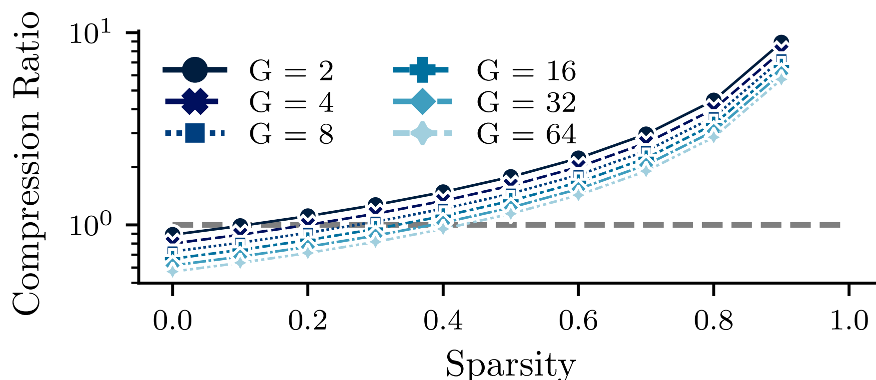


Figure 5.4: Ideal ratio of dense to sparse storage cost for different PE group sizes, and sparsity levels. Gray line shows the break-even point between sparse and dense storage.

However, there is a way of implementing wide SC dot products while maintaining better compression ratios enabled by smaller group sizes. Until now, we have only considered dot products consisting of a single group, referred to as *single-group* sparse SC PEs. Alternatively, we can construct a wide dot product using multiple smaller PEs. For example, a dot product of width $K = 32$ can be constructed using $L = 4$ PEs with $G = 8$ or eight with $G = 4$. We refer to those as *multi-group* sparse SC PEs, where the number of groups L is equal to K/G. An example of single- and throughput-equivalent, multi-group, sparse processing element with L groups is shown in Figure 5.5 a) and b), respectively. They are not strictly equivalent because the sparsity structure required for the multi-group PE is more restrictive - the capacity is now uniformly distributed among individual groups instead of

the whole dot product width.

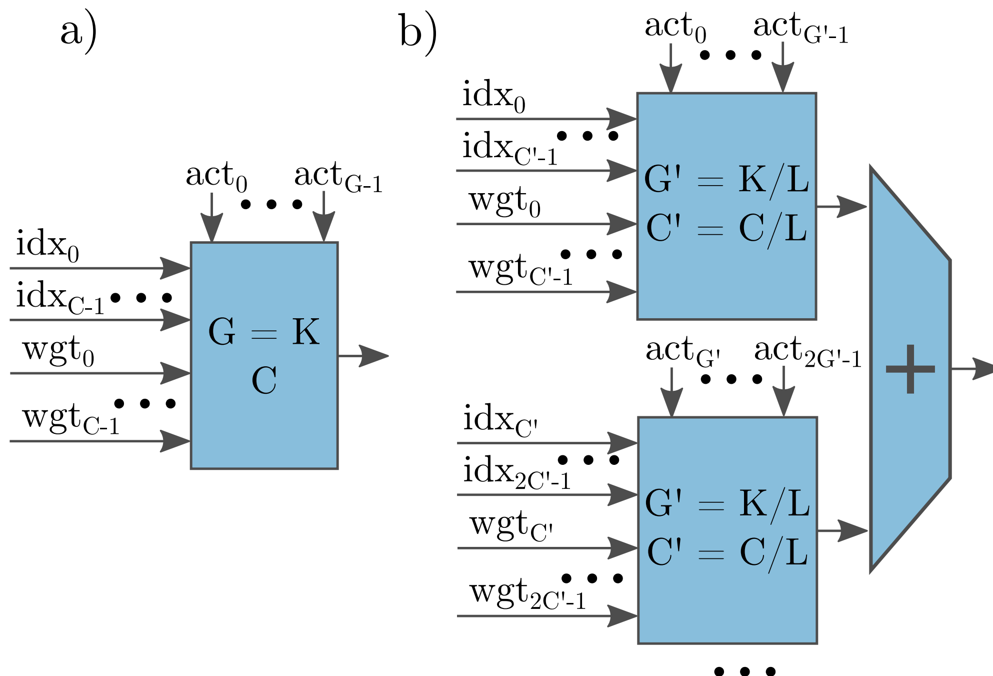


Figure 5.5: Single-group sparse PE with a group size G, capacity C and dot product width K (a), and a throughput-equivalent multi-group sparse PE with L groups.

5.3.4 SASCHA PE Analytical Model

While the sparse SC PE is expected to be more area and energy-efficient than the dense one when weight groups are balanced, i.e., highly sparse, the opposite will happen when running a dense network. In the worst-case scenario, iteration overhead will cause a $\lceil G/C \rceil$ times longer runtime when the same number of dense and sparse PEs is used. To evaluate potential benefits at different sparsity levels, we develop a simple analytical SASCHA PE iteration overhead model. For simplicity, we assume that sparsity is uniformly distributed among weights. The iteration overhead of a multi-group sparse PE of size K, with $L = K/G$ groups, will be determined by the group with the largest number of non-zero weights. Therefore we want to find out the expected maximum number of non-zero values in a group of size G

across L groups. For a group of size G, the probability of having O non-zero weights given sparsity S is:

$$P_{NZ=O} = \binom{G}{O}(1-S)^{O}S^{G-O} \tag{5.1}$$

Where $S \in \langle 0, 1 \rangle$ indicates the ratio of zero weights to all weights. Probability that across L groups of size G, one or more have O non-zero values, and non have more than O:

$$P_{LGO} = \sum_{i=1}^{L} \binom{L}{i} P_{NZ=O}^{i} P_{NZ<O}^{L-i} \tag{5.2}$$

Where $P_{NZ<O}$ is the probability that a group of size G has fewer than O non-zero values:

$$P_{NZ<O} = \sum_{i=1}^{O-1} \binom{G}{i}(1-S)^{i}S^{G-i} \tag{5.3}$$

The average maximum number of non-zero values A across L groups of size G is therefore:

$$A = \sum_{i=0}^{G} i P_{LGi} \tag{5.4}$$

And the expected iteration overhead I, given group capacity C is:

$$I = \frac{A}{C} \tag{5.5}$$

In our model, we assume that dot products where all weights are zero can be skipped entirely in the sparse PE example, as explained in Section 5.4. This behavior allows the iteration overhead to become less than 1. We used the above model to estimate iteration overheads of a multi-group sparse SC PE of width $K = 16$ at different group sizes, capacities, and sparsity levels. Results, normalized by the area, are shown in Figure 5.6 a), with a reference line showing the latency break-even point compared to a dense PE with the same K. It shows that configurations with larger group sizes are not as efficient at low sparsity

levels but much better on highly sparse networks. For example, while sparse PE with $G = 8$ and $C = 1$ is on average 22% slower than the one with $G = 2$ and $C = 1$ at sparsity below 40%, it is on average 63% faster at higher sparsity levels. Further, increasing the capacity is an efficient way of improving the throughput: PE with a group size of 8 and capacity of 4 is, on average, 32% faster than the one with a capacity of 1 when normalized to the area. Based on those results, we will opt for larger group sizes, e.g., 4 and 8, compared to smaller ones.

Figure 5.6 b) compares the iteration overhead obtained through the model with the one obtained using an ideal scheduler described in Section 5.4 on the CIFAR-10 TinyConv network, for a PE with $G = 4$. Our model achieves a 0.996 correlation with the scheduler results, which justifies our choice of modeling weight sparsity using a uniform distribution. While our design could be optimized better towards forms of structured pruning, we want SASCHA to be flexible enough to handle any form of unstructured sparsity without putting the burden on machine learning researchers to conform to the underlying hardware. Using capacity > 1 seems like an obvious choice given its area-throughput benefits. This insight might explain the configuration chosen by [216] since the above analytical model is also applicable to fixed-point PEs. However, we will now discuss an alternative way of improving the throughput of sparse PEs that is unique to SC, and enables yet another design axis to explore.

5.3.5 Parallel Stream Processing

As Figure 5.3 shows, sparse SC PEs can be as much as 2.7X smaller than dense ones. The straightforward way of using this area advantage to increase the throughput is by packing more PEs in the same area. Unfortunately, in the case of sparse SC computation, this approach would yield only limited improvement. As shown in Figure 5.6, depending on the group size and capacity, iteration overheads can be as high as 4 or 8 times. Doubling, or tripling, the number of PEs would not be enough to compensate for it. Another way, as shown

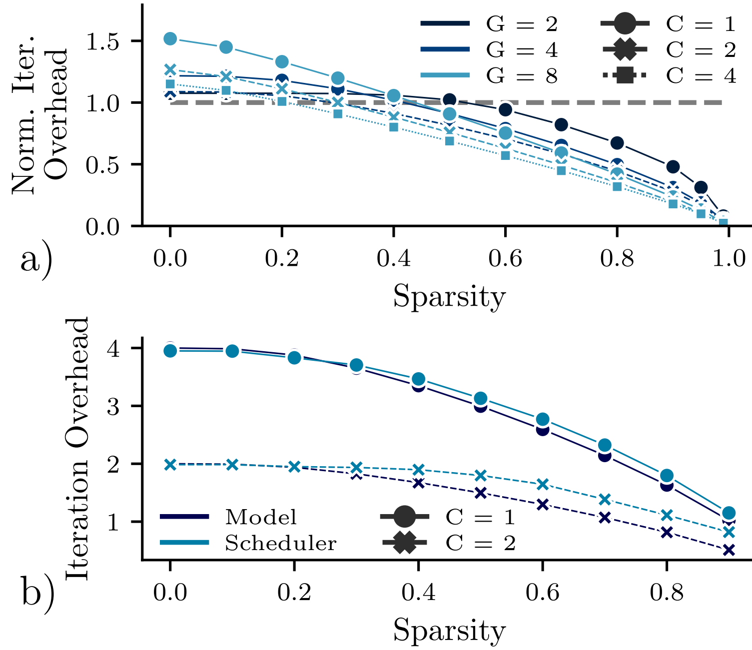


Figure 5.6: Multi-group $K = 16$ sparse SC PE iteration overheads normalized to dense PE area (GEO-style), for different group size G and capacity C, at different sparsity levels, estimated using the analytical model (a). Gray line shows the latency break-even point with a dense PE. Iteration overhead difference between the model and an ideal scheduler described in Section 5.4 on the CIFAR-10 TinyConv network, for a PE with $G = 4$ (b).

in the previous subsection, is to increase the capacity, which shows a good area-throughput trade-off.

However, stochastic computing provides us with another option for increasing the computation throughput. Until now, we assumed that stochastic streams are processed sequentially - one bit at a time. However, by using multiple SNGs, multipliers, and adders per weight, the computation can be parallelized by a varying degree, cutting down the stream processing time and improving throughput [223, 224]. An example of a sparse SC PE with group size G, capacity $C = 1$, and $P = 2$ parallel streams is shown in Figure 5.7. Stream parallelism factor P can be varied to improve the throughput at the cost of additional area. This area-throughput trade-off space is unique to SC and not applicable to conventional fixed- and

floating-point architectures, except for bit-serial ones [219, 19].

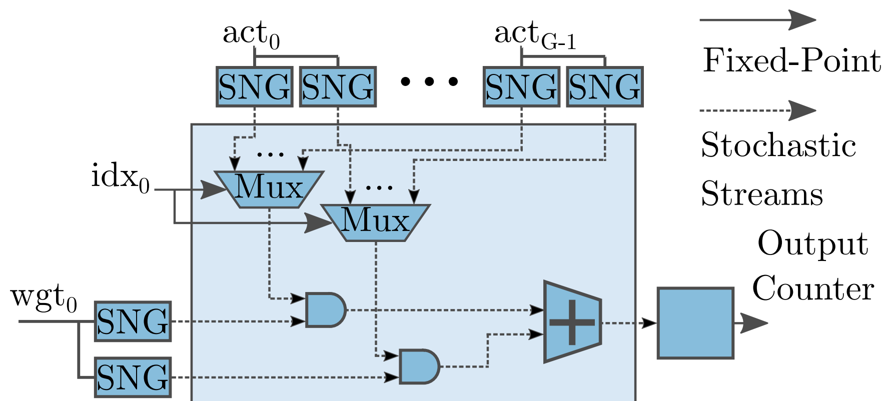


Figure 5.7: Sparse SC PE with group size G, capacity $C = 1$, and $P = 2$ parallel streams. Split-unipolar accumulation fabric is omitted for readability.

If it is possible to apply parallel stream processing to sparse SC PEs to improve their throughput, the same technique could be applied to dense ones. However, the cost of increasing the stream parallelism is much higher for the dense PEs. Figure 5.8 shows the area of a 32x32 array of $K = 32$ PEs for dense and sparse PEs with different group sizes. Capacity is fixed at 1. Buffers, SNGs, and output counters are included. The area of the dense compute grows at a much faster rate with P than the sparse ones because the dense implementation requires many more SNGs, LFSRs, and compute units than sparse ones. For example, the GEO-style array with $G = 2$, $C = 1$, and $P = 2$ is only 5% larger than the dense one with $P = 1$ while providing the same throughput in the dense case. Dense implementation with parallel streams scales especially poorly for the uGEMM-style implementation, where each parallel stream path requires a local decorrelating SNG. Because of that sparse uGEMM-style arrays are significantly smaller, up to 9.5X.

From now on, we will refer to the multi-group, parallel-stream sparse SC PE as *SASCHA PE*. SASCHA PE can be uniquely identified by a set of parameters K, G, C, P, where K is the dot product size, G is the group size, C is the capacity, and P is the stream parallelism factor. We will restrict our evaluation to group sizes of 8 and smaller, which guarantees high

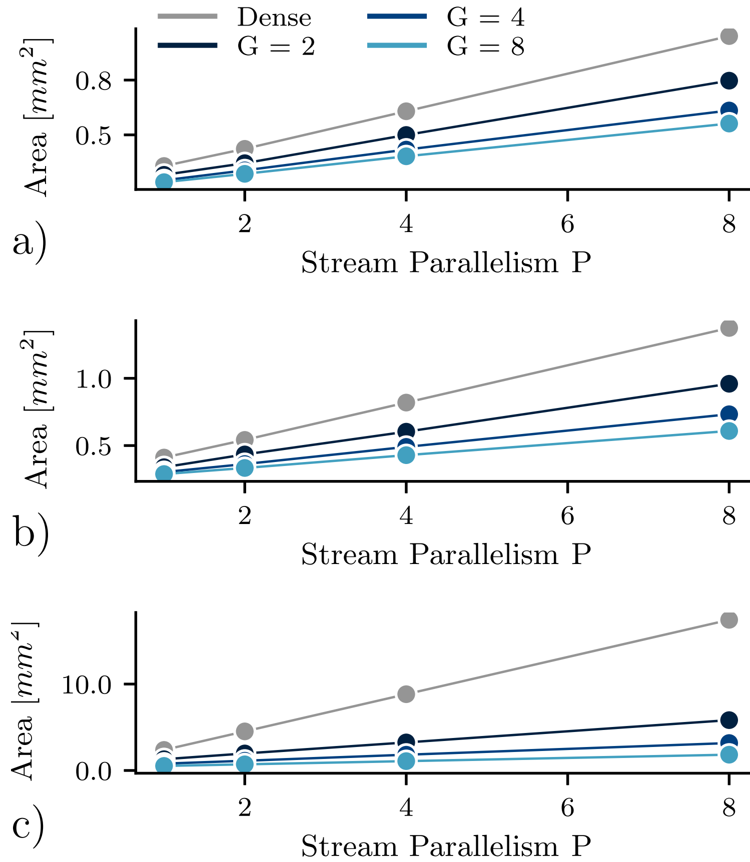


Figure 5.8: Total area of a 32x32 array of $K = 32$ GEO (a), GEO+ (b) and uGEMM (c) PEs, dense and sparse, with different stream parallelism factors. $C = 1$ for all sparse configurations.

storage compression ratios. To further restrict the design space, we will use SASCHA PEs with $PC/G = 1$.

5.4 SASCHA Architecture

5.4.1 SASCHA Accelerator

SASCHA accelerator block diagram is shown in Figure 5.9. It uses a highly parallel PE array similar to [64] and [224]. It relies on broadcasting and spatial data reuse, where all

PEs in the same row share the same set of weights, and all PEs in the same column share the same set of activations. As explained in [64], this structure is uniquely suited for SC computation given small PE sizes and low wire congestion, as opposed to conventional fixed- and floating-point computation. The major distinction in our architecture is the use of SASCHA PEs, instead of dense processing units, using a sparse weight storage format and additional indexing and circuitry required to support asynchronous scheduling, as described below. We refer to the number of rows as M, the number of columns as N, and the dot product width as K. Each of the M rows operates on a set of CK/G weights (WGT), their corresponding indices ($\mathrm{IDX_W}$), and the parent filter index ($\mathrm{IDX_F}$). The latter is needed to support asynchronous scheduling as described in Section 5.4.2. Apart from SASCHA PEs, it contains a weight memory and PCK/G SNGs for parallel stream generation and merger units required for asynchronous scheduling. All N columns share the activation memory, which is organized as a ping-pong buffer [225] to facilitate simultaneous input reads and output writes. Similarly to [10], we use a near-memory vector unit to handle additional partial sum accumulation, batch normalization, scaling, and activation functions. We assume the use of ReLU activation, but the vector unit could be modified to support different ones.

The architecture in [64] focuses on accelerating general matrix multiplication (GEMM) operations, which can be used to implement both fully-connected and convolutional layers in neural networks. Since those two layer types frequently consume $> 90\%$ of inference runtime [226], optimizing the efficiency of GEMM computation is highly desirable. However, while fully-connected layers yield themselves to GEMM representations naturally, convolutional layers need to be transformed. There are two common ways of doing that: image to column (im2col) and kernel to row (kn2row) [227]. In most cases, the latter is desirable, as it does not result in the input replication required by im2col. On the other hand, kn2row can result in compute underutilization on layers with a small number of input channels, Z [227]. For our SASCHA architecture and scheduler, we implement layers that satisfy $Z < K$ using im2col to maintain high utilization, while layers that satisfy $Z \geq K$ using kn2row. In both

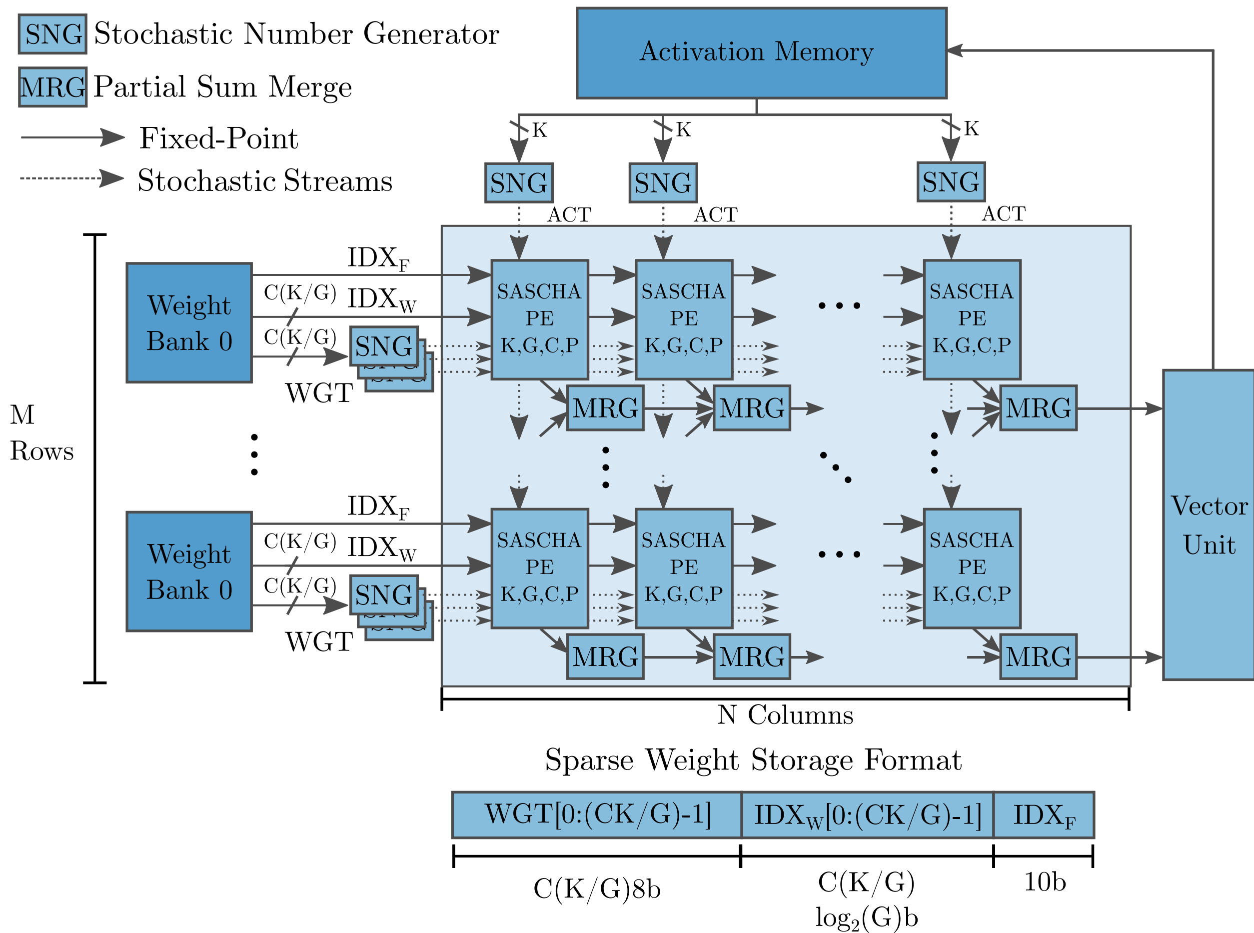


Figure 5.9: SASCHA accelerator architecture block diagram. Partial sum output connections were omitted for readability.

scenarios, filters are partitioned into *partial filters* whose sizes match the dot product width K. Partial filters that come from different *parent filters*, but correspond to the same spatial extents, can be scheduled concurrently in multiple rows of the SASCHA array as they can be multiplied with the same sets of inputs, producing partial sums corresponding to the same row and column in the output tensor.

5.4.2 SASCHA Asynchronous Scheduler

We now discuss the strategy for scheduling computation in a SASCHA architecture defined by M, N, K, G, C, and P parameters. Naively, after performing im2col or kn2row unrolling, we can assign each partial filter to a specific row in the array and co-schedule M partial filters at a time. For the dense architecture, an example schedule of five partial filters, each with $K = 4$, using an architecture with $M = 4$ rows, is shown in Figure 5.10 a). We refer to this as *dense synchronous scheduling* since the execution of each group of M partial filters has to be synchronized. However, in the SASCHA case, as shown in Figure 5.1 c), depending on the level and structure of sparsity, as well as K, G, and C parameters, a given partial filter can be decomposed into a different number of balanced groups. If multiple partial filters are scheduled synchronously, their overall execution time will be constrained by the one with the lowest sparsity, as shown in Figure 5.10 b), for $K = G = 4$, and $C = 1$ referred to as *sparse synchronous scheduling*. In this toy example, synchronous scheduling leads to $I = 2$ iteration overhead (assuming $P = 1$) and 50% compute underutilization.

We propose the SASCHA *sparse asynchronous scheduling* to improve scheduling efficiency. In essence, while the sparse synchronous approach operates on partial filters before decomposition into balanced groups, the SASCHA asynchronous scheduler works with individual balanced groups after decomposition. For all partial filters that correspond to the same spatial subset of original filters, their decomposed balanced groups are combined into a single list. The list elements are then sequentially scheduled onto available rows while keeping track of which partial filter they belonged to initially. The resulting SASCHA asynchronous

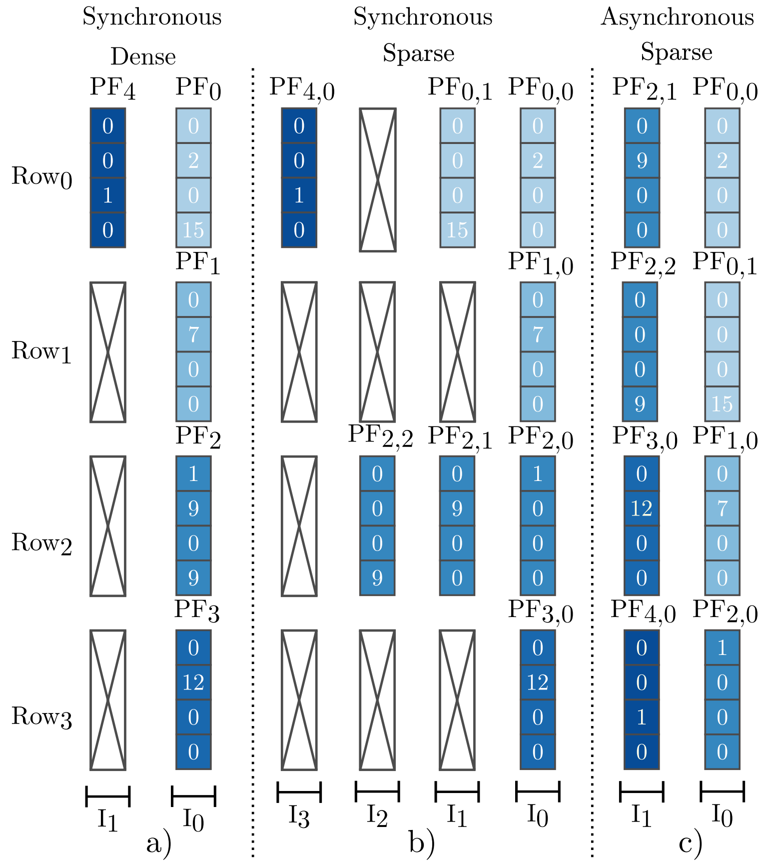


Figure 5.10: Three schedules of 5 partial filters, with $K = G = 4$ and $C = 1$, on an architecture with $M = 4$ rows: dense synchronous (a), sparse synchronous (b) and sparse asynchronous (c). Crossed out boxed indicate compute underutilization.

schedule for the same set of partial filters as before is shown in Figure 5.10, resulting in the same iteration count as the dense schedule and 100% utilization. Using the asynchronous scheduler comes at the cost of additional storage. Each balanced group now needs to carry information about which parent filter it belongs to so that it can be written to the correct location in memory. However, we estimate this penalty to be modest - the worst-case overhead, given $K = 32$ and $G = 8$, would be 30%, while for $G = 2$, it would be below 10%. The resulting compressed weight storage format is shown in Figure 5.9. Weight bank word size will depend on G, K, C parameters, but since those are dictated by PE configuration and

fixed for a given SASCHA implementation, weight memory width can be explicitly provisioned for it. We assume a 10-bit parent filter index, allowing us to index up to 1024 filters, which is enough for commonly used neural network models.

We have implemented both the sparse synchronous and asynchronous schedulers in software. They take as an input a trained network and SASCHA configuration and output iteration counts. The asynchronous scheduler cannot guarantee perfect utilization if the total number of balanced groups corresponding to a set of partial filters is not divisible by M. To assess the effectiveness of the asynchronous scheduler, we also consider the *ideal* scheduler, which is the asynchronous scheduler for a single row that can always be perfectly utilized. Combined results for all three schedulers for the convolutional layers of the CIFAR-10 TinyConv network [95], using a $N = 32, M = 32, K = 32$ SASCHA array with different group sizes are shown in Figure 5.11 a). All configurations have $C = 1$ and $CP/G = 1$ (iso-throughput PEs). When parallel streams are used to compensate for the loss of MAC throughput, larger group sizes are better at converting sparsity into lower iteration overhead. Using a group size of 8 has, on average, 1.3X and 1.6X lower iteration overhead than when using group sizes of 4 and 2, respectively. With $G = 8$, iteration overhead starts decreasing at as low as 10% sparsity, while $G = 4$ and $G = 2$ require sparsity of at least 40% and 70% to start showing benefits. In the best case of $G = 8$, our asynchronous scheduler has on average 1.4X lower iteration overhead than the sparse synchronous one, up to 2.2X at 90% sparsity. It is also, on average, within 11% of the ideal scheduler.

Figure 5.11 b) compares the iteration overhead with different group sizes and capacities while maintaining $CP/G = 1$ when using the asynchronous scheduler. It shows that using parallel stream processing is a more efficient way of using the additional area than increasing the capacity. SASCHA with $G = 8$, $C = 1, P = 8$ is on average 1.2X and 1.46X faster than $C = 2, P = 4$ and $C = 4, P = 2$ configurations, respectively. For group size of 4, $C = 1, P = 4$ is on average 1.15X faster than $C = 2, P = 2$. This conclusion is unique to SC - conventional fixed- and floating-point accelerators do not have access to stream parallelism

design trade-offs. Therefore, we will focus on configurations with $C = 1$ and $P = G$ as the optimal SASCHA PE choices.

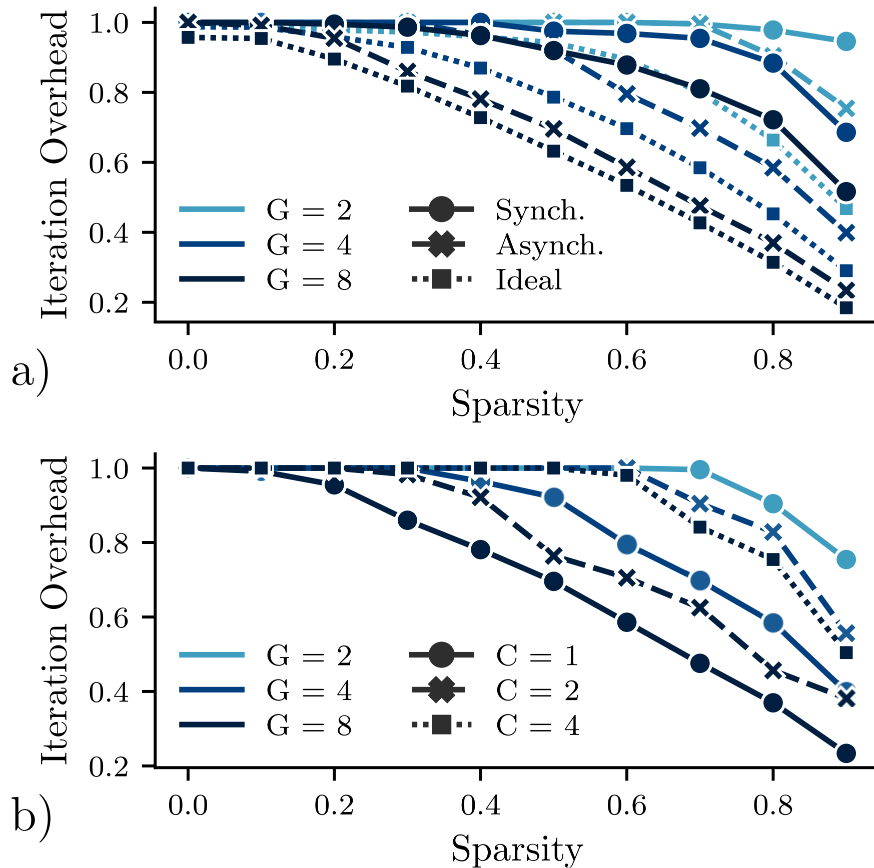


Figure 5.11: Iteration overhead using different sparse scheduling methods (a) and different group sizes and capacity using the sparse asynchronous scheduler (b).

5.4.3 Memory Organization

While beneficial from the point of view of runtime, using the asynchronous scheduler comes at a cost. Synchronous scheduling allows the simple combining of partial sums from different balanced groups corresponding to the same partial filter, as they are assigned sequentially to the same row, as shown in Figure 5.10 b). It means that individual balanced groups do not generate multiple partial sum memory accesses, which can be very costly. For the

asynchronous scheduler, partial filters can now be distributed across multiple rows, making such combining non-trivial. To avoid generating unnecessary memory accesses, we implement merging logic on the datapath used for flushing partial sums out of the PE array, as shown in Figure 5.9. By having parent filter indices ($\mathrm{IDX_F}$) associated with partial sums, those corresponding to the same parent filter can be accumulated when being flushed out of the array and before being written back to activation memory.

We used our scheduler to model the number of memory accesses for activations, weights, and partial sums, for dense and sparse architectures using different schedulers and dataflows. Results of the convolutional layers of the CIFAR-10 TinyConv network, at 90% sparsity, are shown in Figure 5.12. Output stationary dataflow is impossible when using the asynchronous scheduler due to balanced groups belonging to the same parent filter being potentially distributed across different PE units. Input stationary dataflow is, therefore, the best choice of the dataflow for the SASCHA asynchronous scheduler, cutting down the number of memory accesses by 18% compared to weight stationary at high sparsity levels. It is also within 7% and 13% of the best achievable dataflow for dense and synchronous scheduling, respectively.

5.5 Bit-Slicing Weights

While SASCHA architecture, discussed in the previous Section, shows high latency improvements on sparse networks, it can, at most, maintain the same throughput when running dense ones. In this Section, we show how higher effective sparsity and more efficient hardware can be extracted by exploiting intra-value sparsity of unpruned weights through *bit-slicing*. By bit-slicing, we mean decomposing weights into smaller slices and processing them individually, then scaling the results depending on the LSB position of a given slice. For example, two-way slicing involves splitting the fixed-point weight value into equally sized MSB and LSB slices, multiplying them individually with each corresponding activation, scaling the MSB result, and adding both. A similar technique has been proposed in the context of SC

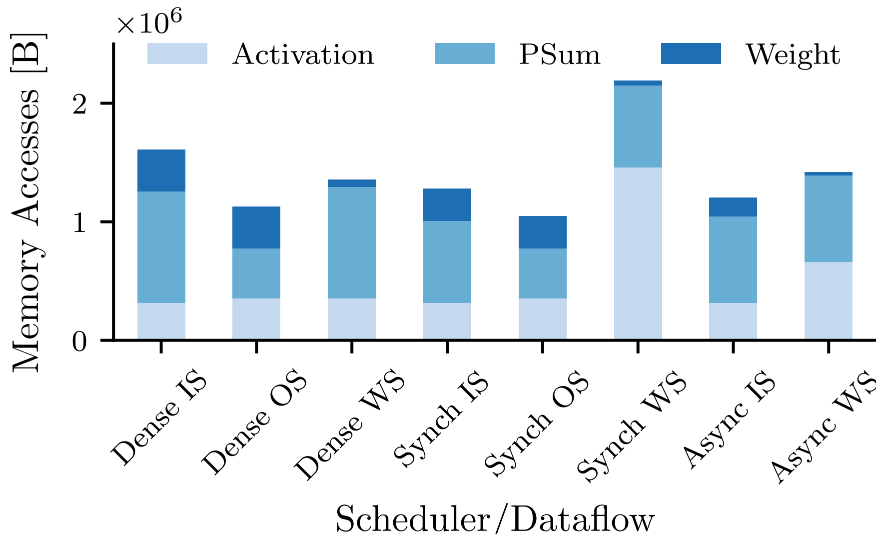


Figure 5.12: Number of memory accesses in bytes for the convolutional layers of CIFAR-10 TinyConv network at 90% sparsity, depending on the choice of scheduling and dataflow. All results for $M = 32, N = 32$, and $K = 32$. Sparse results for $G = 8, C = 1$, and $P = 8$.

in [224]. However, it is used only as a means of reducing the computation stream length and not exploiting additional operand sparsity. For SASCHA, we assume an equal split between the number of most significant and least significant bits. While other split sizes and granularities are possible, their analysis is beyond the scope of this work.

The idea behind improving sparsity with bit-slicing comes from the observation that if we divide a set of values into bit-slices, the resulting sparsity, i.e., the percentage of slices that are completely zero, will be at worst the same and at best higher than sparsity of non-sliced values. Given that weights in neural networks exhibit zero-centered bell-shaped distributions, we would expect the sparsity of MSB bit-slices to be even higher. To verify this, we evaluated the overall, MSB, and LSB sparsity in the convolutional layers of the CIFAR-10 TinyConv network at different network pruning levels. Results are shown in Figure 5.13 a). We can see that even for the unpruned networks, MSB slices exhibit very high sparsity - 64%. While the high MSB sparsity could help when processing dense networks, we expect the benefits to be minuscule at high sparsity levels - the MSB and LSB sparsity of the network pruned

to 90% is 90% and 90.1% respectively, meaning there is not a lot of additional computation savings available.

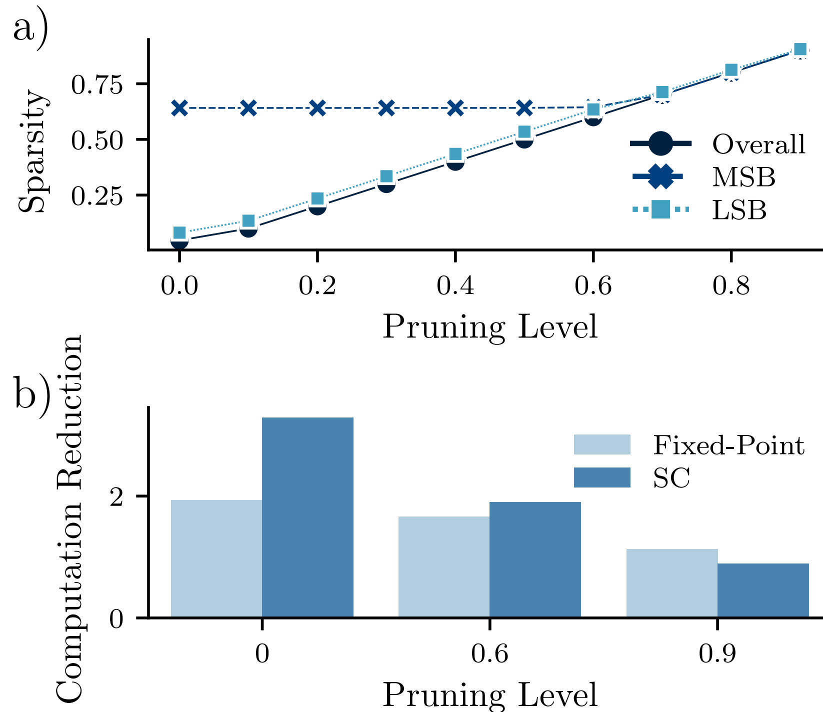


Figure 5.13: Overall, MSB and LSB sparsity for (a) and reduction in sliced multiplication area x delay cost relative to non-sliced cost for SC and fixed-point (b), at different pruning levels for CIFAR-10 TinyConv.

We assume that only one of the operands, weight, is sliced, as we mainly care about extracting more sparsity on the weight side. In a naive implementation, where each slice is computed with the native stream length, e.g., 64, this would lead to a minuscule runtime reduction at best since most of the LSB slices are not sparse. However, we can capitalize on the fact that the MSB slice contribution to computation will be much higher than the LSB one. By computing the MSB part with the original stream length, e.g., 64, and the LSB part with a shorter one, e.g., 8, and then scaling the LSB result and adding it to the MSB one, we can approximate the original result. We refer to this as asymmetric-stream slicing,

as opposed to prior works which used symmetric stream lengths [224]. The comparison of non-sliced and sliced unipolar multiplication is shown schematically in Figure 5.14 a) and b), respectively. As can be seen, when using sliced operands, the size of the SNG and its buffers can be reduced, which improves the area.

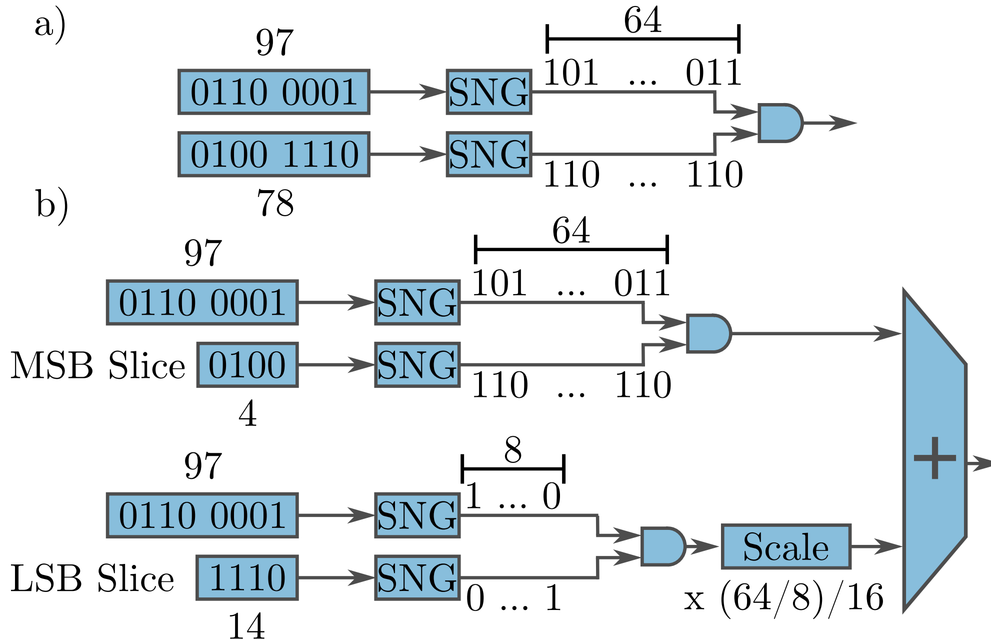


Figure 5.14: SC unipolar multiplication a), and sliced multiplication b).

The slicing technique is also applicable to fixed-point computation. However, while in the fixed-point case it would result in the same precision used for both the MSB and LSB parts, SC makes asymmetric precision possible. In terms of the area-delay product cost of a multiplier, 4-bit MSB/LSB slicing results in a 2.24X reduction for each of the parts compared to a regular multiplier. Assuming 64-bit long MSB streams and 8-bit long LSB streams in the SC case, there is no reduction in the cost for the MSB part, but there is an 8X cost reduction in the LSB part. When considering the sparsity levels of each part in Figure 5.13 a), we see that at low pruning levels the number of LSB slice multiplications dominate, and we expect the asymmetric SC precision to give us higher benefits than symmetric fixed-point slicing. To evaluate this hypothesis, we multiplied the proportion of non-zero MSB and LSB slices

by their area-delay cost reduction factors when slicing and combined the results to show an overall ideal reduction in multiplication cost. Results, normalized to the ideal non-sliced sparsity computation reduction, are shown in Figure 5.13 b). For low and moderate levels of pruning SC slicing achieves higher cost reduction than fixed-point one - 2.6X on average, compared to 1.8X, respectively. While not as effective at high sparsity levels, asymmetric precision of slicing allows us to show improvement even at low sparsity levels, as shown in Section 5.6.

There is a concern about how slicing will affect computation precision. To evaluate that, we analyzed the root mean square error (RMSE) of stochastic unipolar multiplication in both the non-slicing and slicing scenarios using the same stream lengths as shown in Figure 5.14. The operands are drawn from activation and weight distributions of the CIFAR-10 TinyConv model, where weight distribution is used for the sliced operand. The average RMSE across 1000 trials is shown in Table 5.1, and the sliced multiplication error is within 30% of the non-sliced one. We will discuss network-level accuracy and performance impact of slicing in Section 5.6.

Table 5.1: RMSE of unipolar multiplication with and without bit-slicing, w.r.t. floating-point precision, for different stream lengths (1000 trials). LSB stream length is 8.

	Stream Length	16	32	64	128	256
RMSE	No Slicing	4.49	3.14	2.26	1.531	0.98
	With Slicing	4.97	3.28	2.78	1.867	1.31

To summarize, bit-slicing allows us to expose higher levels of sparsity present in weights to SASCHA compute. By utilizing the asymmetric sparsity of MSB and LSB slices, lower relative precision required for the latter, and SC's unique precision-latency trade-off space, bit-slicing enables SASCHA to show performance improvements even on dense networks, as shown in Section 5.6. Bit-slicing can be handled natively by SASCHA at the cost of

underutilizing the sparse storage and SNG buffers (provisioned for 8-bit values). However, for completeness' sake, we also evaluate the SASCHA-S variant, which is dedicated to weight-sliced networks, by having reduced weight storage and SNG buffers. From the scheduler's point of view, each sliced part can be treated as a separate layer. The outputs of those layers are then scaled and added element-wise.

5.6 Evaluation & Results

5.6.1 SASCHA Accuracy

All models are trained using PyTorch. The training setup is similar to the one used in [10], but with added layer-wise magnitude-based pruning. TinyConv, VGG-11 and VGG-16 [1] are trained on the CIFAR-10 dataset, and ResNet-18 and -34 are trained on ImageNet dataset. The fully-connected layers of VGG-16 are reduced to 512 to accommodate the small CIFAR-10 dataset. All models are trained with 64-bit streams. Bit-slicing has little effect on accuracy on VGG-11, and the accuracy with and without bit-slicing differs by less than 0.7%, as shown in Figure 5.15.

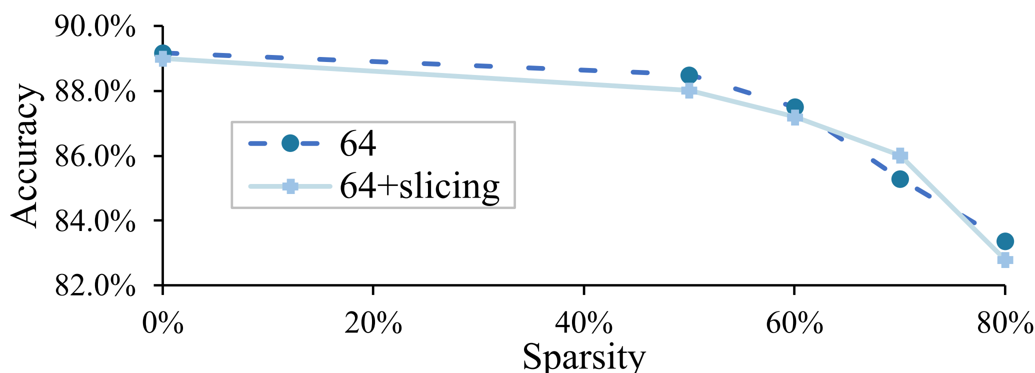


Figure 5.15: Accuracy of CIFAR-10 VGG-11 with different sparsity levels. 0% sparsity means no sparsity constraint.

Figure 5.16 summarizes the results on CIFAR-10. Compared to GEO and ACOUSTIC,

which also use OR accumulation, accuracy has been improved by switching the order of ReLU, pooling, and batch normalization (bn). While previous works use pooling-bn-ReLU order to achieve spatial pooling, SASCHA does not have the same constraint and can use the more optimal bn-ReLU-pooling. This change improves accuracy by 2% for both TinyConv and VGG-16 on CIFAR-10. Due to its small size, TinyConv is less resilient to sparsity. At 60% sparsity, accuracy drops by 0.3-0.8%. Accuracy drop using VGG-16 is milder, with no noticeable drop in accuracy when using 60% sparsity and $\approx 4.5\%$ using 90% sparsity. While slicing reduces accuracy when the models are dense, the accuracy gap reduces with increased sparsity. While the gap is 1-1.7% for the dense models, the gap is negligible or even reverses at maximal sparsity usable for each model.

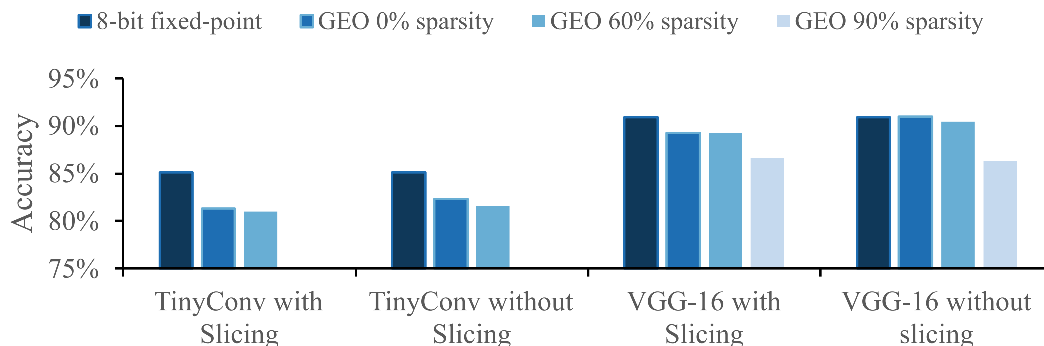


Figure 5.16: SASCHA CIFAR-10 Top-1 accuracy with dense and sparse networks.

Figure 5.17 summarizes the results of Resnet-18 and Resnet-34 on ImageNet. We use a higher accuracy version of stochastic computing, denoted as GEO+. In GEO+, full binary accumulation replaces partial binary accumulation, eliminating OR accumulation. Since only inputs within the same OR accumulation window require different seeds, all multiplications can use the same seed pair. We further improve the multiplication accuracy by choosing the seed pair that produces the lowest error. With this modification, SASCHA achieves comparable accuracy to 8-bit fixed-point throughout different sparsity levels. Because uGEMM achieves accuracy comparable to 8-bit fixed-point without retraining, as reported by [64], we

expect it will behave similarly with pruning. Since no efficient stream simulation functions are available for uGEMM, we have not trained it separately.

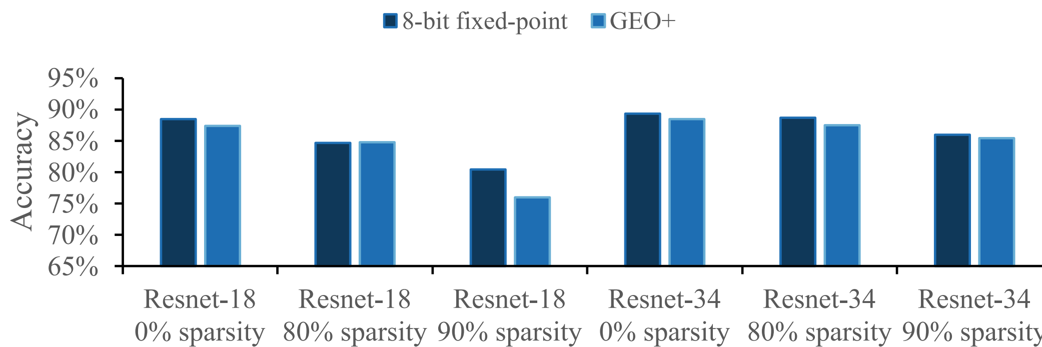


Figure 5.17: SASCHA ImageNet Top-5 accuracy with dense and sparse networks.

5.6.2 Performance Results

To evaluate SASCHA performance, we implement the entire GEMM array, including individual components such as SASCHA PEs, SNGs, LFSRs, merge blocks, vector unit, and the necessary glue logic using Verilog HDL, and synthesize them using TSMC 28nm library and Cadence Genus synthesis tool, at 400MHz clock frequency. We use the results to estimate overall area, power, and energy consumption of different SASCHA configurations. We use the scheduler described in Section 5.4 to estimate runtime and memory accesses required for different networks and levels of sparsity. We use the CACTI tool to estimate the cost of memory accesses [177]. To demonstrate how SASCHA is agnostic to the underlying style of SC computation, we evaluate three configurations: SASCHA-GEO, -GEO+, and -uGEMM, using the PEs as described in Section 5.3.

We compare SASCHA to GEO ULP [10], which uses the same underlying computation as SASCHA-GEO but is optimized towards convolutional layers. We use the same simulator as described in [10] to estimate performance. Also, for each of the SASCHA versions, we compare it with a corresponding dense version, with the same number of PEs, and no

stream parallelism, referred to as GEMM-GEO, -GEO+, and -uGEMM. The last one is very similar to the original uGEMM architecture [64] but uses binary instead of streaming accumulation. To keep the results consistent with uGEMM, we only report the logic area without including memories. For a comparison with sparse fixed-point accelerators, we use SCNN [79] and Laconic [122]. For a fair comparison with SASCHA, we omitted the area of on-chip memories, based on the area breakdowns provided by the original works, and scaled the area of compute and buffers to account for the change in precision from 16 to 8 bits. We also scaled the technology node to 28nm using the scaling equations provided in [214]. We then configured the number of each accelerator's PEs to roughly match the area of SASCHA. We refer to those configurations as SCNN-M and Laconic-M, respectively. Their execution is modeled using DNNSim [228]. We omit ResNet-34 results on Laconic-M, as the simulator was not able to schedule the computation successfully. All designs are iso-frequency.

Based on the results discussed in the previous sections, we limited our exploration to configurations with $G = 4$ or 8, as they are better at extracting sparsity benefits than $G = 2$. We also limit ourselves to configurations with $C = 1$ and $G = P$, since they provide a better area-throughput trade-off than ones with $C > 1$ and lower parallelism. This choice, enabled by parallel stream processing unique to SC, allowed us to arrive at a different design point that would be optimal for fixed-point PEs, as exemplified by [216]. Based on area estimates, for the SASCHA-GEO and -GEO+ versions we picked $M = 32, N = 16, K = 32, G = 4, C = 1, G = 4$, referred to as *SASCHA-GEO* and *SASCHA-GEO+ 4/1/4* as they are within 8% of the logic area of GEO ULP. We also include SASCHA-GEO and -GEO+ configurations with the same M, N, K, and C, but with G and P equal to 8, referred to as *SASCHA-GEO* and *SASCHA GEO+ 8/1/8*. Those configurations, while consuming only 23-42% larger area than the 4/1/4 configurations, are much better at extracting sparsity benefits, as we will show shortly. Figure 5.18 shows the area and power breakdown of SASCHA-GEO 8/1/8, based on individual module synthesis, showing a more balanced, and not SNG-dominated, distribution compared to [10]. SASCHA achieves this through the combination of GEMM-

style architecture and sparsity-oriented design. For the uGEMM variant, due to a much larger PE area, we size the array with $M = 16, N = 16, K = 16$, for both 4/1/4 and 8/1/8 versions, which brings them close to the iso-area with the GEO and GEO+ variants. Finally, we include two SASCHA-GEO and GEO+ configurations with the same parameters as the ones above, but with bit-slicing support, referred to as *SASCHA-GEO-S* and *SASCHA-GEO+-S*. We do not include SASCHA-uGEMM slicing configurations, as the impact of slicing on the uGEMM-style PE accuracy is beyond the scope of this work. All non-slicing configurations use a stream length of 64, while the slicing ones use 64-bit long streams for the MSB computation and 8-bit long streams for LSB. We assume that memory bandwidth is provisioned for maximum expected throughput. When reporting sparse results, we pick the maximum sparsity at which SC accuracy is within 4% of fixed-point, GEO for CIFAR-10, and GEO+ for ImageNet, sliced or non-sliced, whichever is higher.

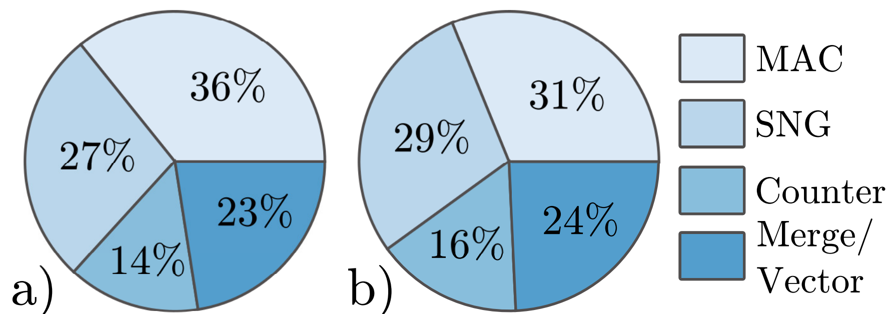


Figure 5.18: Area (a) and power (b) breakdown of SASCHA GEO 8/1/8.

Final results are shown in Table 5.2, for the TinyCONV and VGG-11 and -16 networks on the CIFAR-10 dataset, and ResNet-18 and -34 networks on the ImageNet dataset. Results are shown at two different levels of sparsity. We first analyze the performance of dense networks. Compared to the dense GEMM versions with the same array size, non-slicing SASCHA configurations maintain a similar throughput while suffering at most 31% loss in energy efficiency. This is expected - while stream parallelism is used to recover runtime, it lowers PE energy efficiency through lower SNG reuse. Further, in the dense case, SASCHA will require more overall memory accesses due to indexing overheads and asynchronous scheduling. The

Table 5.2: Area [mm^2], power [mW], throughput [Fr/s] and energy-efficiency [Fr/J] for different accelerators, models and datasets, and sparsity.

			CIFAR-10 TinyConv				CIFAR-10 VGG-11				CIFAR-10 VGG-16				ImageNet ResNet-18				ImageNet ResNet-34			
			Sparsity 0%		60%		Sparsity 0%		70%		Sparsity 0%		90%		Sparsity 0%		80%		Sparsity 0%		90%	
	Area	Power	Fr/s	Fr/J	Fr/s	Fr/J	Fr/s	Fr/J	Fr/s	Fr/J	Fr/s	Fr/J	Fr/s	Fr/J	Fr/s	Fr/J	Fr/s	Fr/J	Fr/s	Fr/J	Fr/s	Fr/J
Architecture	[mm2]	[mW]	[k]	[k]	[k]	[k]	[k]	[k]	[k]	[k]	[k]	[k]	[k]	[k]								
SCNN-M	0.3	-	1.7	-	3.0	-	0.11	-	0.42	-	0.11	-	0.42	-	8	-	79	-	4	-	46	-
Laconic-M	0.2	-	3.4	-	3.6	-	0.07	-	0.07	-	0.12	-	0.12	-	46	-	49	-	-	-	-	-
GEO ULP	0.24	50	10.6	240	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
GEMM-GEO	0.20	78	6.6	72	-	-	0.5	5.5	-	-	0.26	2.9	-	-	50	499	-	-	27	299	-	-
SASCHA GEO 4/1/4	0.21	75	6.7	73.5	8.8	93.5	0.8	8.2	1.1	10.5	0.65	7.0	0.95	9.8	50	525	105	969	27	301	76	754
SASCHA GEO 8/1/8	0.30	93	6.7	60.2	12.2	102.7	0.9	7.2	1.5	11.1	0.81	7.0	1.39	11.2	50	435	151	1078	27	246	121	921
SASCHA-GEO-S 4/1/4	0.19	62	6.8	76.2	7.5	83.7	1.3	10.8	1.5	11.9	0.68	7.3	0.84	8.4	46	481	97	821	29	357	68	736
SASCHA-GEO-S 8/1/8	0.26	80	8.8	73.6	10.3	86.1	1.6	10.1	2.1	11.6	0.91	7.3	1.21	8.9	47	435	140	863	36	331	108	894
GEMM-GEO+	0.24	81	6.6	70	-	-	0.5	5.3	-	-	0.26	2.8	-	-	50	485	-	-	27	289	-	-
SASCHA GEO+ 4/1/4	0.23	78	6.7	71.0	8.8	90.4	0.8	8.0	1.1	10.2	0.65	6.8	0.95	9.5	50	508	105	941	27	290	76	731
SASCHA GEO+ 8/1/8	0.32	99	6.7	56.1	12.2	96.1	0.9	6.7	1.5	10.5	0.81	6.5	1.39	10.5	50	406	151	1019	27	229	121	867
SASCHA-GEO+-S 4/1/4	0.21	65	6.8	75.4	7.5	82.8	1.3	10.7	1.5	11.8	0.68	7.2	0.84	8.3	46	476	97	814	29	353	68	755
SASCHA-GEO+-S 8/1/8	0.27	83	8.8	72.5	10.3	84.8	1.6	10.0	2.1	11.4	0.91	7.2	1.21	8.8	47	375	140	854	36	325	108	881
GEMM-uGEMM	0.34	112	2.1	17.5	-	-	0.1	1.0	-	-	0.06	0.55	-	-	13	106	-	-	7	57	-	-
SASCHA uGEMM 4/1/4	0.26	82	2.0	23.0	2.9	32.5	0.2	2.3	0.3	3.4	0.19	2.1	0.36	3.9	13	148	33	354	7	77	26	272
SASCHA uGEMM 8/1/8	0.27	86	2.0	21.7	4.0	40.8	0.2	2.2	0.4	4.1	0.23	2.4	0.57	5.5	13	142	48	475	7	72	42	399

exception is SASCHA-uGEMM, where going from dense to sparse-parallel PEs can reduce area and power, due to the large size of the former, which results in marginally higher energy efficiency in the dense case. In the case of CIFAR-10 VGG networks, where the network has high *natural* weight sparsity without pruning, throughput, and energy efficiency are improved by up to 3.6X and 4.4X, respectively. Bit-slicing SASCHA-S configurations perform much better on unpruned networks, as expected. By exploiting high inherent MSB slice sparsity, they can improve the throughput by up to 1.33X over GEMM Dense and improve energy efficiency by up to 1.2X, except in the CIFAR-10 VGG case, where improvements are higher. GEO achieves higher throughput and energy efficiency on the dense TinyConv, which comes from the fact that its architecture is highly optimized toward convolutional layers. Compared to SCNN-M, SASCHA-GEO, and GEO+ can improve the throughput by 4X-8.7X, owing to the higher efficiency of SC compute over fixed-point. SASCHA-uGEMM configurations, despite having 4x fewer PEs than the other configurations, still outperform SCNN-M by 2.2X. Compared to Laconic-M, GEO and GEO+ configurations have up to 19X, while the

uGEMM ones have up to 7.8X speedup on dense networks.

When running moderately sparse (60%) TinyConv, SASCHA accelerators improve the throughput by up to 1.92X compared to the dense variants and up to 1.94X compared to the unpruned network on the respective SASCHA configurations. At 90% sparsity, SASCHA configurations can be up to 6.5X faster and 5.5X more energy-efficient than the dense versions. Compared to their respective versions running dense networks, they improve runtime by up to 8.8X and energy efficiency by up to 10.1X. SASCHA-GEO and GEO+ maintain a 1.5X to 4X throughput advantage over SCNN-M on sparse convolutional networks, despite SCNN taking advantage of both weight and activation sparsity, while SASCHA only utilizes the former. SASCHA-uGEMM 8/1/8 outperforms SCNN-M by up to 2.2X on convolutional networks. While sliced configurations are not as efficient at high sparsity, they still achieve up to 4.7X and 3.4X throughput improvement over GEMM Dense and SCNN-M, respectively, on sparse convolutional networks. Laconic-M extracts most of its benefits on a *bit-sparsity* level even without pruning and does not show large improvements when small weights are removed.

In Table 5.3 we show the achieved weight compression ratio for all four evaluated SASCHA configurations and three evaluated networks. For weight compression, the underlying PE architecture does not matter. As expected, at high sparsity, bit-slicing configurations have lower effective compression due to indexing overhead affecting both the LSB and MSB slices. The exception is the VGG-11 network, where a combination of high natural sparsity and relatively low pruned sparsity allows the slicing configurations to come out ahead. SASCHA 8/1/8 has a higher compression ratio at 90% sparsity compared to 4/1/4, despite higher indexing overhead. This is due to more efficient weight storage - for the same dot product width, it will store half of the weights in each memory word compared to SASCHA 4/1/4. For the latter, at high sparsity, many of those words will be underutilized.

Table 5.3: Weight compression ratio for different SASCHA configurations, networks, and sparsity levels.

Model	TinyConv	VGG-11	VGG-16	ResNet-18	ResNet-34
Sparsity	60%	70%	90%	80%	90%
SASCHA 4/1/4	1.09	2.00	4.22	1.43	2.11
SASCHA 8/1/8	1.22	2.27	5.34	1.75	2.85
SASCHA-S 4/1/4	0.90	2.34	4.07	1.22	1.68
SASCHA-S 8/1/8	0.93	2.52	5.19	1.38	2.10

5.7 Related Work

5.7.1 Sparse Accelerators.

Exploiting sparsity to improve performance in hardware has been extensively studied for floating- and fixed-point accelerators. Some of the prior works only try to exploit the sparsity of one operand type, like Cnvlutin (activations) [229], or Cambricon-X (weights) [90]. Others, like Cambricon-S [230], SCNN [79], Bit-Tactical [219], or TensorDash [218], can exploit both activation and weight sparsity, often through a combination of static and dynamic scheduling. While most accelerators opt for some form of operand advancing through a staging window [218, 219, 217], others like SCNN [79] or MatRaptor [222] rely on multiplying all non-zero operands and mapping the results to appropriate partial sums afterward. Due to the high cost of detecting and supporting sparse execution, the majority of sparse accelerators focus on higher precision arithmetic like 32-bit and 16-bit floating-point or 16-bit fixed-point [90, 218, 79, 217, 222]. Such datatypes and accelerators are more suited to training neural networks. In contrast, SASCHA focuses on approximate edge inference, where quantized, 8-bit, and lower fixed-point precision has become a standard [20]. Despite only focusing on weight sparsity, it can outperform fixed-point accelerators that also exploit sparse activations,

thanks to highly efficient stochastic computation.

5.7.2 Stochastic Computing Accelerators.

While stochastic computing has been enjoying a recent renaissance due to its synergies with deep learning algorithms; there is a surprising lack of configurable, system-level designs available. A few examples include ACOUSTIC [5], which is an accelerator targeting convolutional neural networks specifically, GEO [10], which improves on ACOUSTIC's accuracy and performance, uGEMM [64] and StoRM [224], which are flexible general matrix multiply engines, and SCOPE [166], and in-memory DRAM accelerator. We compare SASCHA with GEO and uGEMM, which target similar, low-precision edge inference. StoRM can be considered a specific case of uGEMM with specialized PEs. While it explores operand slicing, it processes them in a spatially unrolled manner, requiring symmetric stream lengths and not being able to utilize additional sparsity exposed by it, unlike SASCHA. SCOPE is a data center accelerator with area requirements orders of magnitude higher than SASCHA. Other works, like HEIF [105], or SC-DCNN [106] have proposed generating custom hardware for specific neural network models. Those approaches often lead to impractically high area and are of limited utility in the rapidly changing neural network landscape. BISC-MVM [107], and SkippyNN [104], propose more accurate stochastic multiplier designs but are limited to fixed-point addition and do not present system-level elaboration. Finally, some recent works propose methods of doing stochastic computing in a deterministic manner, without introducing any error [187]. However, they often require long stream lengths for processing and are not competitive in terms of latency and energy.

5.8 Conclusion

In this Chapter, we presented SASCHA - a sparsity-aware neural network accelerator architecture using stochastic computing. SASCHA exploits sparsity in a way that synergizes

with the main advantages of SC. It encompasses a sparse multiply-accumulate block design, GEMM accelerator architecture, and asynchronous scheduling method. Further, we propose a bit-slicing method unique to SC that can exploit sub-operand sparsity even in dense networks. At 90% weight sparsity, SASCHA can be up to 6.5X faster and 5.5X more energy-efficient than comparable dense SC accelerators with a similar area, and up to 8.7X faster than sparse fixed-point accelerators, without sacrificing performance on dense networks. Our future work will explore the interplay between sparsity and bitstream length in the context of SC.

CHAPTER 6

SCIMITAR: Event-Based Tracking with Stochastic Compute-In-Memory

Event-based cameras are a relatively new class of devices that offer low latency and bandwidth, high dynamic range, sparse imaging data that is well suited towards high-speed object tracking. However, this sparse format is not compatible traditional architectures used in computer vision, geared towards dense, frame-based information. There is a need for new devices that can work efficiently with event-based data, taking advantage of its inherent sparsity, while being able to take advantage of its low latency. In this Chapter, we propose SCIMITAR: Stochastic Computing In-Memory In-situ Tracking ARchitecture for Event-Based Cameras, an accelerator for high-speed object tracking. SCIMITAR uses stochastic compute-in-memory (SCIM) for highly efficient analog processing, in-situ stream generation for compact implementation, and a host of optimizations for utilizing input sparsity. SCIMITAR provides unparalleled throughput for ROI-based processing, with both latency and energy that can scale with sparsity. We demonstrate SCIMITAR performance on an object tracking application using detailed circuit level simulations.

Collaborators:

- Jiyue Yang, Electrical and Computer Engineering, UCLA.
- Alexander Graening, Electrical and Computer Engineering, UCLA.
- Vinod Kurian Jacob, Electrical and Computer Engineering, UCLA.

- Professor Sudhakar Pamarti, Electrical and Computer Engineering, UCLA.
- Professor Puneet Gupta, Electrical and Computer Engineering, UCLA.

6.1 Introduction

A new class of imaging devices has emerged in recent years - the so-called event-based cameras [231]. In contrast to their conventional, frame-based counterparts, those cameras do not capture the entire image at a fixed frame rate. Instead, they try to imitate the behavior of a biological retina by transmitting only the information about changes in their field of view as an asynchronous event stream [231]. Certain characteristics of event-based cameras, like implicit background filtering, low latency, and high dynamic range, make them a potential replacement for frame-based ones for some classes of applications. One of those applications is object tracking, where event-based cameras are already showing a great promise [232, 231, 233, 234]. However, the developing nature of this field means that researchers are still establishing the best way of processing event-based data. Approaches vary from purely asynchronous ones [235, 236, 237, 238], to ones that try reconstructing frames from events [239, 240, 241, 242, 243]. While former are not yet well established, the latter do not take full advantage of the sparse nature of the event stream.

At the same time, the low-latency, high data rate nature of event-based cameras proves to be a blessing and a curse. On the one hand, it makes it possible to track high-velocity objects without motion blur plaguing conventional cameras [240]. On the other hand, general-purpose architectures cannot deal with the requirements of low-latency and sparsity presented by this type of data. One solution to this problem is using domain-specific accelerators, custom-made for a particular application [12]. Indeed, multiple prior works have proposed custom accelerators for event-based data, using FPGAs [235, 244], or ASICs [72].

To push the performance of event-based object tracking systems, we propose SCIMITAR:

Stochastic Computing In-Memory In-situ Tracking ARchitecture for Event-Based Cameras. SCIMITAR uses two techniques that have become popular in machine learning accelerators in recent years - stochastic computing (SC) [5, 64, 104], and compute in-memory (CIM) [166, 245, 246]. Both are apt at processing low-precision, linear algebra kernels with low latency and high energy-efficiency. What is more important, they can be combined as a stochastic compute in-memory (SCIM) accelerator, which makes it possible to further improve the performance by removing costly analog-digital converters (ADCs) [245]. SCIM is a good candidate for processing event-based data, although it comes with a set of challenges. First, it requires spatial unrolling of stochastic streams, which can lead to large chip area [245]. Second, it cannot easily take advantage of the sparse nature of the event stream, leading to many inconsequential computations. SCIMITAR solves those issues through the use of: in-situ generation, which alleviates the need for stream unrolling, and a host of microarchitectural optimizations utilizing sparsity.

Our contributions are as follows:

- First event-based object detection and tracking accelerator using in-situ stochastic computing in-memory (SCIM) processing.
- Evaluation of different event-based data processing methods from the point of view of hardware efficiency.
- In-situ stream generation for SCIM architectures, which can significantly cut down the area requirement of SCIMITAR.
- A set of micro-architecture techniques to support high input sparsity, that demonstrably improve the energy efficiency of SCIMITAR.

6.2 Motivation

6.2.1 Event-Based Cameras

Event-based cameras are sometimes referred to as neuromorphic cameras since the original design and data format was inspired by the biological retina [231, 247]. The biological retina is thought to send pulses to the brain as individual cells in the eye sense changes in intensity. Similar to this, event-based cameras only transmit pulses when the brightness value of a specific pixel changes. Output from an event camera is typically a stream of positive and negative events depending on whether a change is an increase or decrease in brightness. Pixels that do not change do not send events. In a situation where the camera is stationary, this means moving objects will cause events, but stationary background objects will not, thus highlighting the most important information in the scene for many applications such as tracking [231, 248, 249, 250]. Figure 6.1 shows how moving objects are highlighted while the background disappears. If the event-based camera is moving, it will highlight the edges of objects in its field of view, providing useful data for SLAM-type applications [231, 251, 252, 251, 253].

Event-based cameras have several benefits compared to traditional frame-based cameras, especially when it comes to object tracking:

- *Sparse Data Output*: Since only events are transmitted, in most applications, only a small portion of the pixels will be active at a time. This behavior compares to frame cameras, where all pixels transmit the measured brightness for every image/frame captured. [231, 247]
- *Fast Response to Changes*: Event-based cameras can have very high maximum firing rates for pixels, and they are not constrained by data output rate as much as frame cameras due to the sparse format. For existing cameras, pixel latency varies from 120 to 3 μs. [231, 254, 255].

- *High Dynamic Range*: Event-based cameras have very high dynamic range due to the individual behavior of pixels (avoiding a shared exposure level) and because the implementations for event-based cameras typically respond to changes in the log of intensity instead of operating on a linear scale. This can give a dynamic range of 130-140dB for event cameras compared to a dynamic range around 60dB for a traditional camera. [231, 254, 256]
- *Very High Temporal Resolution*: Event-based cameras can provide continuous information about the location of fast-moving objects that might move a lot between frames on a traditional camera. Existing cameras support event rates between 1 and 1200 Meps (mega-events per second) [231, 254, 257].

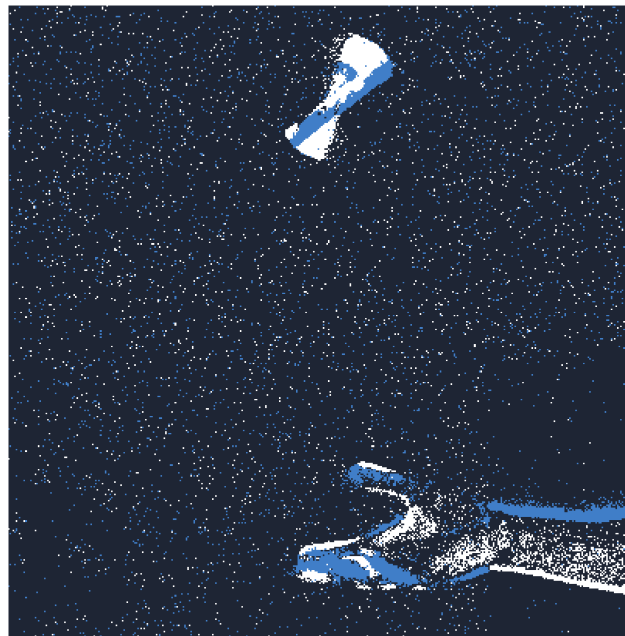


Figure 6.1: Spinning Marker. This image shows event data generated from tossing a spinning whiteboard marker into the air with a cluttered background. The stationary background has disappeared so it is easy to see the moving objects. The white events are positive events indicating an increase in brightness and the blue events signal a decrease in brightness.

6.2.2 Event-Based Data Processing

Given this new data representation, we have to determine the best way of processing such information. While it ultimately depends on the application, most approaches can be categorized into three groups [231]. First are the algorithms that process individual events [235, 236, 237, 238]. Those *event-based* methods update their state on every incoming event, guaranteeing minimum latency while avoiding the processing of irrelevant data. However, given the young age of this field, such techniques are still not well developed. The second type of approach processes events in groups [239, 240, 241, 242, 243]. It often involves reconstructing the image frames. The frame format makes it possible to employ an abundance of well-established computer vision (CV) algorithms. However, frame reconstruction causes the sparsity of event data to be lost - frames will mostly be filled with inactive, zero-valued pixels. We will refer to those methods as *frame-based* algorithms. Finally, the last type uses so-called *patches* or *regions-of-interest* (ROIs) [258, 259, 260, 261]. This can be considered a compromise between the first two categories - while still reconstructing partial frame information, only pertinent fragments of it (ROIs) are processed. Processing ROIs makes it possible to avoid processing large, inconsequential parts of the scene while still being able to harness conventional CV methods. Such *ROI-based* algorithms have been shown to drastically improve performance compared to frame-based ones [262], by reducing the amount of computation required. The three approaches are shown schematically in Figure 6.2.

In this work, our goal is to find an approach that best complements custom hardware, to provide the fastest and most energy-efficient implementation for tracking applications. For this purpose, we need to consider not just the cost of computation, but also memory, which is frequently overlooked, despite having a potentially much higher impact on system performance [12]. In order to gain intuition into how different event data processing techniques compare, we build a simple analytical performance model. We assume that a camera has an HxW resolution and M events are processed in a single iteration. The overall interval in which the events are acquired (t) is divided into D bins, and events belonging to a particular

bin end up in the corresponding time channel. This spatiotemporal *voxel grid* representation is a commonly used way of reconstructing event data, as it preserves temporal information [231, 240]. As a proxy for computation, we assume the use of N spatiotemporal filters of the size $KxKxD$, convolved with the image data for object detection, either the reconstructed frame/ROI or individual events. Such filters, for example, Gabor ones, are commonly used for object detection in tracking applications [263], and we use them as a general proxy for computation cost. The maximum of a given filter is recorded and used for tracking, for example, by Kalman or particle filters [259, 260]. For ROI-based processing, we assume that C ROIs of size $RxRxD$, where $R << W, H$, are processed. We omit the computational cost of selecting ROIs in this analysis. Prior works have shown that ROI detection, or *region proposal*, requires much lower energy and latency compared to filtering itself [264, 265, 262]. All parameters of our model are shown in Figure 6.2.

For each approach, we consider the performance metrics shown in Table 6.1. In all cases, we make the most optimistic assumptions about hardware capabilities. First, we are interested in memory access counts. For simplicity, we assume all accesses are of the same size. For all cases, the number of weight accesses is the same and equals filter size (KKD) times the number of filters (N). In the case of event-based processing, only $2M$ input accesses are required - one write and read per event. However, assuming that events are processed sequentially, each convolution result would need to be individually updated, leading to M event updates of KK convolution results for N filters. Given that there is no trivial way of predicting where within the frame incoming events are located, there is also no way of collating the results on the fly to reduce the number of output accesses without resorting to frame- or ROI-based processing.

For frame-based processing, we require M event writes, and a frame read of size WHD. However, assuming the filtering is completely unrolled spatially, all results are available, and all maxima can be determined at the same time, leading to only N output writes. Similarly, ROI-based processing requires M event writes, and C ROI reads of size RRD. Maxima

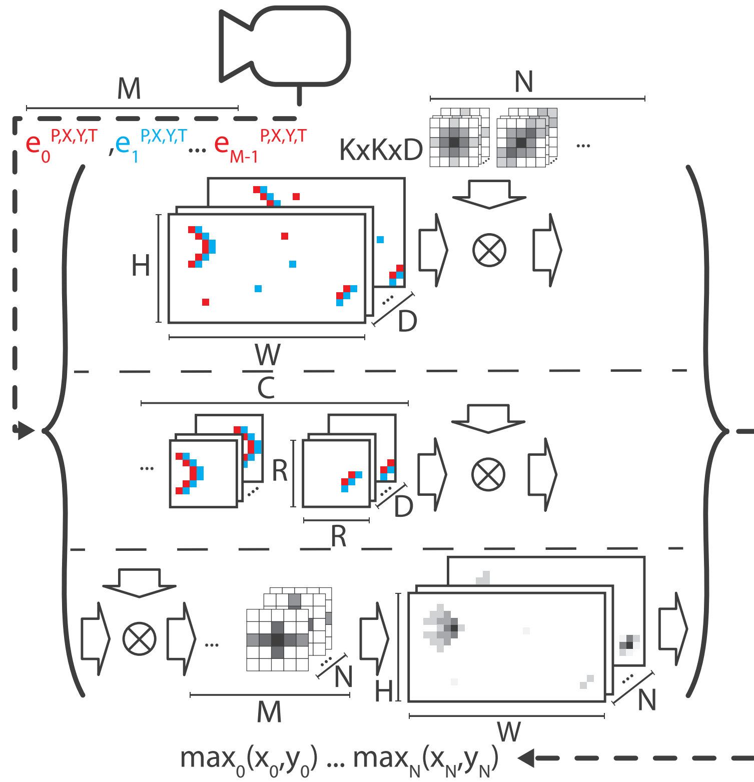


Figure 6.2: Analytical model parameters, for frame- (top), ROI- (middle), and event-based (bottom) processing.

Table 6.1: Analytical event-based tracking performance model. *Event*, *frame*, and *ROI* refer to event-, frame-, and ROI-based processing, respectively.

		Event	Frame	ROI
Accesses	Input	2M	M+WHD	M+RRDC
	Output	KKMN	N	CN
	Weight	KKDN	KKDN	KKDN
MAC		KKMN	KKDWHN	KKDRRCN
Reuse	Input	KKN	KKN	KKN
	Output	1	WH	RRC
	Weight	1	KKD	KKD

need to be updated on every ROI, leading to CN output writes. We measure the required computation in multiply-accumulate (MAC) operations. For event-based processing, M events need to be convolved with N filters of size KK. For frame-based processing, the entire WHD frame needs to be convolved with N filters of size KKD. For ROI-based processing, C ROIs of size RRD need to be convolved with N filters of size KKD.

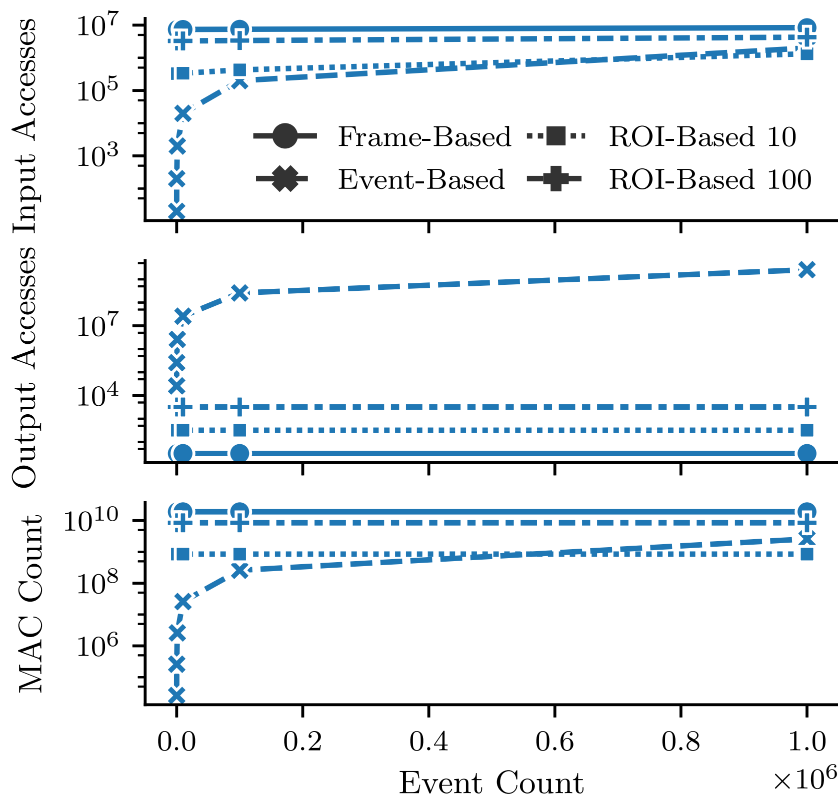


Figure 6.3: Input (top) and output (middle) memory accesses, and MAC count (bottom), with varying event count.

Let us now assume a camera resolution of $W = 1280, H = 720$ pixels, based on commercially available models [231]. We then assume $D = 8$ time step channels. For filters, we assume $N = 32$, $K = 9$ spatio-temporal Gabor filters. For ROI-based processing, we assume $R = 64$ region size. Figure 6.2 shows how memory access counts and MAC change when the number of events being processed varies from 10 to 1 million. We consider two

ROI-based cases, with 10 and 100 ROIs, respectively. We see that while with a low event count, event-based processing requires significantly fewer input accesses and computation than other methods, that efficiency is quickly lost when the event count increases. At the 100k event mark, event-based processing is comparable to ROI-based one with 100 ROIs. More importantly, the number of output accesses is significantly higher than other methods, even at relatively low event counts.

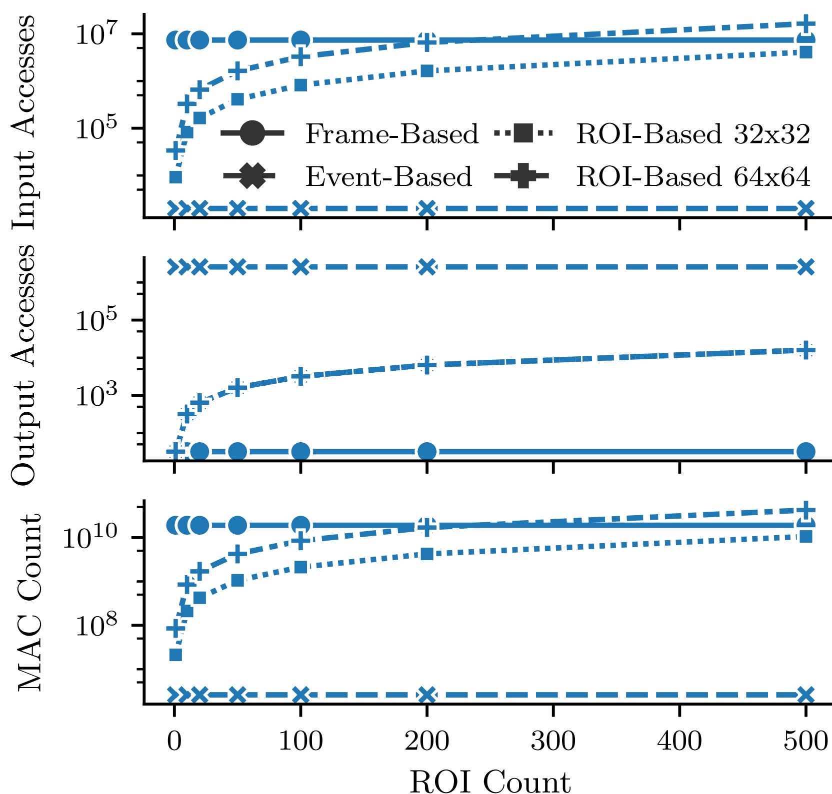


Figure 6.4: Input memory accesses (top), and MAC count (bottom), with varying ROI count.

We then compared the results for a fixed event count ($M = 1000$) but with a varying ROI count and size ($R = 64$, and $R = 32$). Results are shown in Figure 6.4. We see that if the number of ROIs can be kept low, ROI-based processing provides a good trade-off between event- and frame-based processing. Finally, we also consider possible data reuse in each of the approaches, as it can be used to amortize both memory and computation cost, especially

in custom hardware architectures [13]. As shown in Table 6.1, while all approaches can exploit the same level of input reuse, event-based processing cannot easily take advantage of weight or output reuse. Based on all of the above, we decided to focus on ROI-based processing: it allows us to take advantage of *parallelism* and *data reuse* while providing a good *efficiency trade-off* between event- and frame-based approaches.

6.2.3 Stochastic Computing In-Memory

Given the focus on 2D convolution operators and ROI-based processing, we are looking for technologies that make it possible to implement large, dense linear algebra kernels in a fast and efficient manner. Fortunately, given the recent emergence of deep neural networks, which rely heavily on linear algebra, there is now an abundance of techniques used for accelerating such computations [13, 246, 5]. This further motivates our choice of ROI-based processing, as purely event-based methods, while available, are not yet as well developed [235, 72]. Stochastic computing, discussed extensively in prior Chapters, is one of such techniques. However, people have now found ways of improving its efficiency even more, in a scheme called Stochastic Compute-In-Memory (SCIM) [245].

SCIM accelerators achieve much higher energy-efficiency by embedding SC's computation logic inside the memory [166, 245, 266]. Conventional Compute-In-Memory (CIM) accelerators perform analog compute inside the memory, which requires power-hungry ADC/DACs [267, 246]. Stochastic Computing (SC) uses tiny digital logic gates such as AND/OR gate as the basic computation unit. Since SC is a digital computing scheme, no ADC/DAC is needed. Figure 6.5 shows the structure of a 256-element MAC unit in an SRAM-based SCIM macro. Each 6T SRAM cell stores a stochastic bit of the weight parameters. Two extra NMOS transistors are added next to the 6T SRAM cell to perform an AND operation between the stored stochastic bit and the compute word line. The outputs of all SCIM cells are connected to form the compute line (CL), which effectively performs a wired-OR operation. When all the multiplication results are zero, the CL stays at VDD. If at least

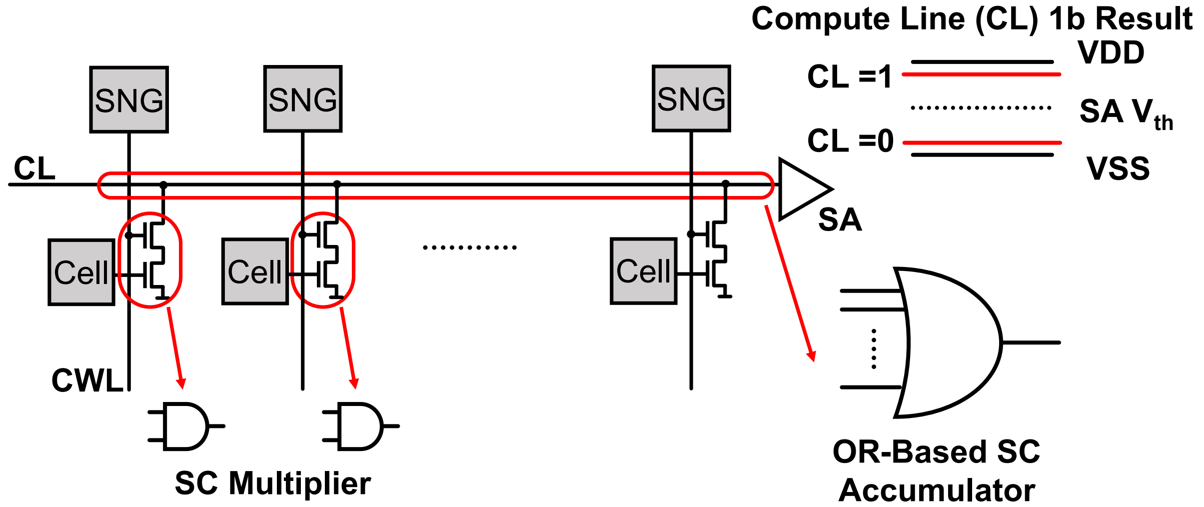


Figure 6.5: 256-wide SCIM MAC structure.

one multiplication result is one, the CL will be discharged to VSS. Due to the large voltage separation between VSS and VDD, a small inverter is used as a sense amplifier, which is robust against PVT variations and local mismatches.

The SCIM macro stores the stochastic representation of the weight parameters, which requires a large amount of storage and an extra conversion step while loading the macro. In the loading cycle, the weight binary numbers are converted to stochastic bits by the column's stochastic number generator(SNG) and written into the macro. Each SCIM macro stores a 1-bit representation of the filters and computes 1 bit for the outputs. 2^N memory cells are needed to accurately represent N-bit precision. For example, a 4-bit binary number requires 16 cells to store stochastic bits in 16 SCIM macros separately. Due to the exponential scaling of the SC's stream length, the area penalty is worse for higher precision compared to conventional CIM solution, which only needs N cells for storage [245].

Based on the above discussion, we believe that SCIM is very well suited towards custom hardware tracking acceleration of event data. However, it comes with a host of issues. First, stream unrolling, as explained above, leads to large area requirements, which can prohibitive, particularly for edge devices [245]. Further, SC and SCIM, with their high

parallelism and spatial reuse, are naturally geared towards dense computation and cannot easily take advantage of event data sparsity [5, 64, 245]. While a SC architecture dealing with *weight* sparsity has been proposed recently [11], here we are mainly concerned with *input* sparsity. In the next section, we explain how our design addresses those issues.

As a note, ROI processing is not unique to event-based data. A similar approach can be applied to conventional cameras, using forms of motion estimation to propose regions of interest. However, this process is computationally cheaper for event-based cameras, as motion estimation is inherently "built into" the data format, and people have proposed that event-based region proposal can be done with very little overhead [231, 264].

6.3 SCIMITAR Implementation

6.3.1 Stochastic Compute-in-Memory Macro with In-Situ SNG

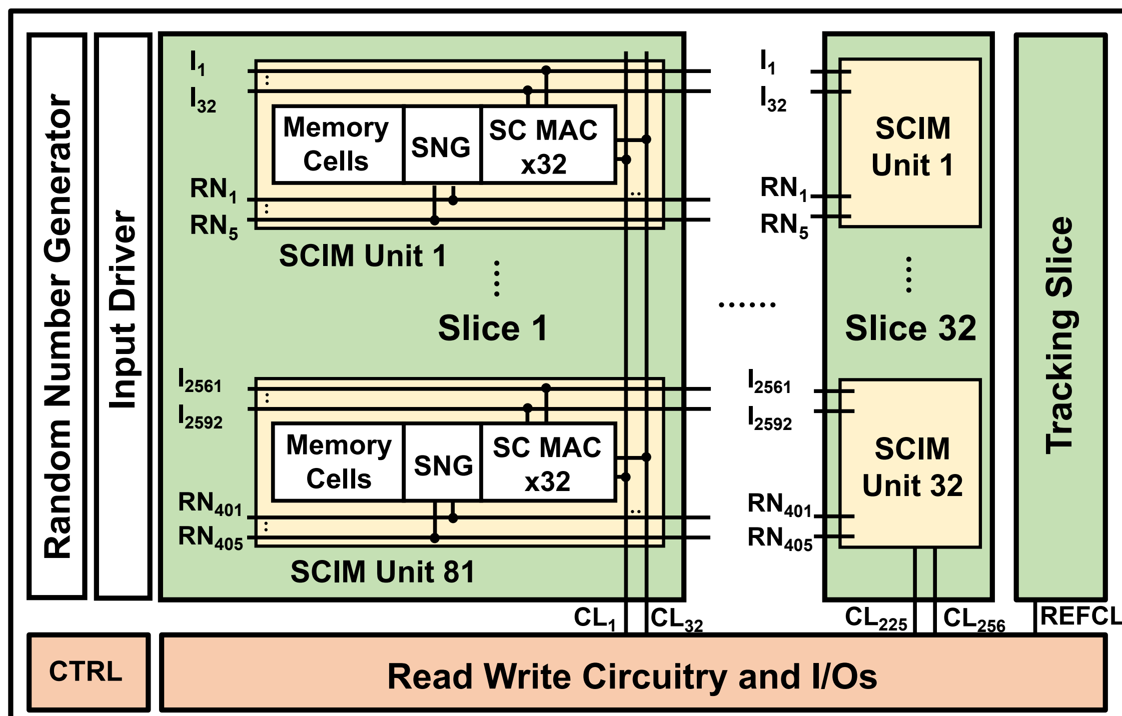


Figure 6.6: SCIM Macro architecture.

In this work, we propose a Stochastic Compute-In-Memory accelerator with an in-situ Stochastic Number Generator (SNG). The in-situ binary to stochastic number conversion allows the SCIM macro to store the binary numbers instead of the unrolled stochastic bits, therefore significantly increasing the macro density. The compact in-situ SNG only has a small area overhead. The output of the in-situ SNG is reused by many in-memory SC MAC units, which further reduces the SNG cost. Figure 6.6 shows an overview of the SCIM macro.

The SCIM unit is the smallest compute primitive within the macro. The SRAM-based memory cell array stores a 6-bit fixed-point weight that has 1 bit for sign and 5 bits for magnitude. The in-situ SNG converts the weight from binary to stochastic number using a 5-bit random number ($RN_1 - RN_5$). The random numbers are generated from the pseudo-random number generator block, next to the memory and shared across the SCIM units on the same row. The SNG output within each SCIM unit is multiplied with 32 inputs ($I_1 - I_{32}$) and accumulated with other SCIM units in the same SCIM slice. Each SCIM slice stores a 9x9 filter and computes 32 outputs, using nine unrolled input half-rows (9x36, including overlap). There are 32 SCIM slices within the macro that share the same inputs, each implementing a different filter. The outputs of the SCIM MAC are streams of 1-bit values and are converted to digital bits by the sense amplifier at each Compute Line (CL). An extra replica slice is built to track the process, voltage, and temperature (PVT) variation of the circuit and generate the timing signals for the output sampling. The following sections will provide a detailed description of the in-situ SNG and the in-memory SC MAC units.

6.3.2 In-Situ Stochastic Number Generator

We propose to embed a compact stochastic number generator (SNG) inside the memory to achieve in-situ binary to stochastic number conversion, which significantly reduces the required storage size. Previous bit-parallel stochastic-CIM macro stores 2^N stochastic number in the memory for N-bit precision [245]. By embedding in-situ SNGs, only N number of cells are required. During computation, the in-situ SNGs use random numbers generated

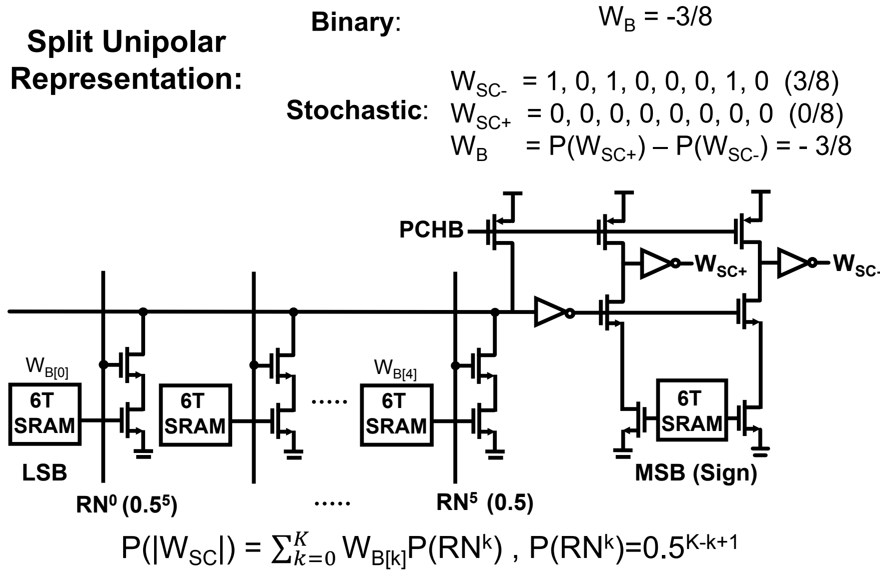


Figure 6.7: Split-unipolar stochastic representation (top) and in-situ stochastic number generator (SNG) circuit (bottom).

near the memory and shared across columns to serially convert the binary numbers stored in the memory to stochastic bits. The in-situ SNG circuit is shown in Figure 6.6. The SRAM memory cells store the binary weight numbers. Besides the sign bit (MSB), each memory cell has two extra cascaded NMOS transistors beside the regular 6T SRAM to perform an AND operation between the stored binary bit and a random number that is generated outside of the array. The probability of a random number is binary weighted from MSBs to LSBs: the leading bit is selected by the random number with the probability of 0.5^1, while the nth LSB is selected with the probability of 0.5^K. The output of AND logic in each cell is connected to form a local bitline, which performs a wired-OR operation. An inverter amplifies in-situ SNG's local bit line and inverts the signal to maintain the correct logic. Using the in-situ SNG also means that the implementation is now agnostic to the stream length being used, which was not the case for bit-parallel SCIM [245].

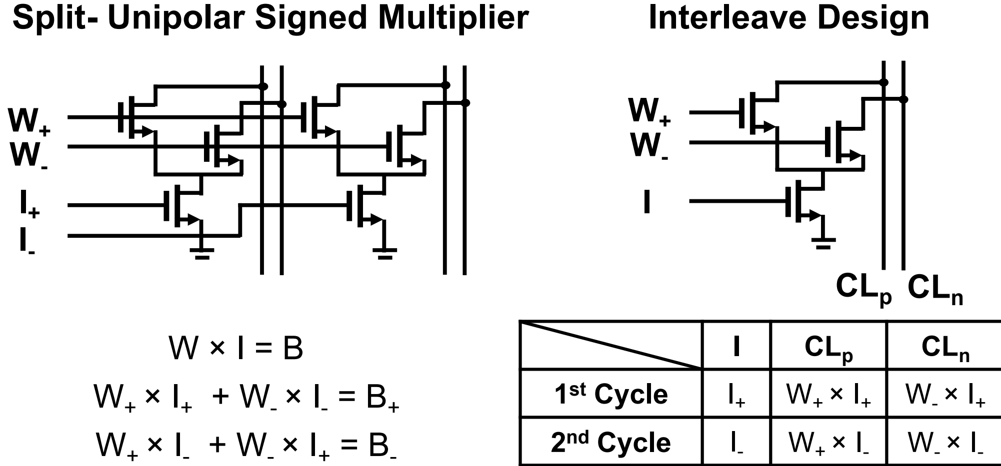
A split unipolar stochastic number generation is used to support signed numbers [5]. In the split-unipolar representation, a signed number is represented by two stochastic bit

streams: W_{SC+} and W_{SC-}, while only one of the bit streams is enabled by the sign bit. The value of the number is encoded as the difference between two streams: $W_{SC+} - W_{SC-}$. If the number is positive, W_{SC+} represents the value's amplitude, and W_{SC-} will produce a stream of zeros, and vice versa for the negative numbers. A demultiplexer circuit is implemented using pseudo-NMOS logic to generate a stochastic bit stream in split-unipolar format. The sign bit stored in a memory cell selects which of the W_{SC+} and W_{SC-} should be kept as zero and passes the SNG output to the other stream. An inverter is added at the output as a buffer.

6.3.3 In-Memory Stochastic MAC Unit

Previous work has demonstrated a stochastic in-memory compute macro for unsigned numbers [245]. In this work, we propose an in-memory SC MAC unit that supports both unsigned and signed computation. With an interleaved multiplier design, higher energy efficiency and lower area overhead of the MAC unit are achieved. Figure 6.8 shows the comparison between a conventional signed SC multiplier for split-unipolar stochastic representation [5], and the proposed interleaved SC multiplier design. The weight is stored in the memory and converted to a split-unipolar stochastic stream(W_+, W_-) by the in-situ SNG. The input is read from a buffer, converted to stochastic bits (I_+, I_-), and applied to the in-memory MAC unit. For the non-interleaved design, input/weight each has two stochastic streams and a cross multiplication is performed: $I_+ \times W_+$, $I_+ \times W_-$, $I_- \times W_+$, and $I_- \times W_-$. Partial output streams with the same sign, e.g. $I_+ \times W_+$ and $I_- \times W_-$ are combined, and then the negative result is subtracted from the positive one to obtain the final result. Individual multiplication is done by an in-memory AND gate using only two NMOS transistors. The output is precharged to VDD. If and only if both operands are high, there is a conducting path pulling down the output to VSS, effectively performing an AND operation.

The interleaved multiplier cell design is used to reduce area overhead and energy consumption. The computation is split into two phases. In the first one, the input's positive



	I	CL_p	CL_n
1st Cycle	I_+	$W_+ \times I_+$	$W_- \times I_+$
2nd Cycle	I_-	$W_+ \times I_-$	$W_- \times I_-$

Figure 6.8: Interleaved Signed SC MAC unit.

streams are applied and multiplied with weights. The intermediate outputs are converted to binary numbers by counters. In the second phase, the input's negative streams are applied. The final result is the addition of the results from two cycles. The interleaved SC multiplier design reduces the area overhead by half, at the cost of processing latency. However, the overall energy consumption is reduced, since a more compact cell area benefits from shorter routing and correspondingly lower parasitics.

Each SCIM unit has 32 SC MACs sharing the in-situ SNG's outputs to increase the weight reuse factor, effectively performing multiplication between one weight and 32 inputs. The large reuse factor significantly amortizes the weight memory access cost and energy overhead of the in-situ SNG. Increasing the reuse factor for the conventional charge-based CIM solution is challenging due to the large area of the ADCs. Each MAC output of the CIM macro requires an ADC. To achieve accurate MAC results, the compute cell using a transistor or metal capacitor requires a large area to improve the matching property. This limits the weight reuse factor of the conventional CIM macro to 1. The SCIM's extremely compact MAC unit only uses three NMOS transistors. Since the SCIM's MAC output is only 1 bit, a simple inverter can be used as a sense amplifier. Figure 6.9 shows the design of a SCIM slice. 32 input lines are routed in the horizontal direction on top of the SRAM cells.

The input lines use minimal spacing and occupy two SRAM cells' height. The 32 inputs multiply with the output of the in-situ SNG by the 32 SCIM MAC units. 81 SCIM units form a SCIM slice that performs 32 81-way dot products. The MAC outputs (CLp1-32 and CLn1-32) are amplified by an inverter at the bottom of the macro.

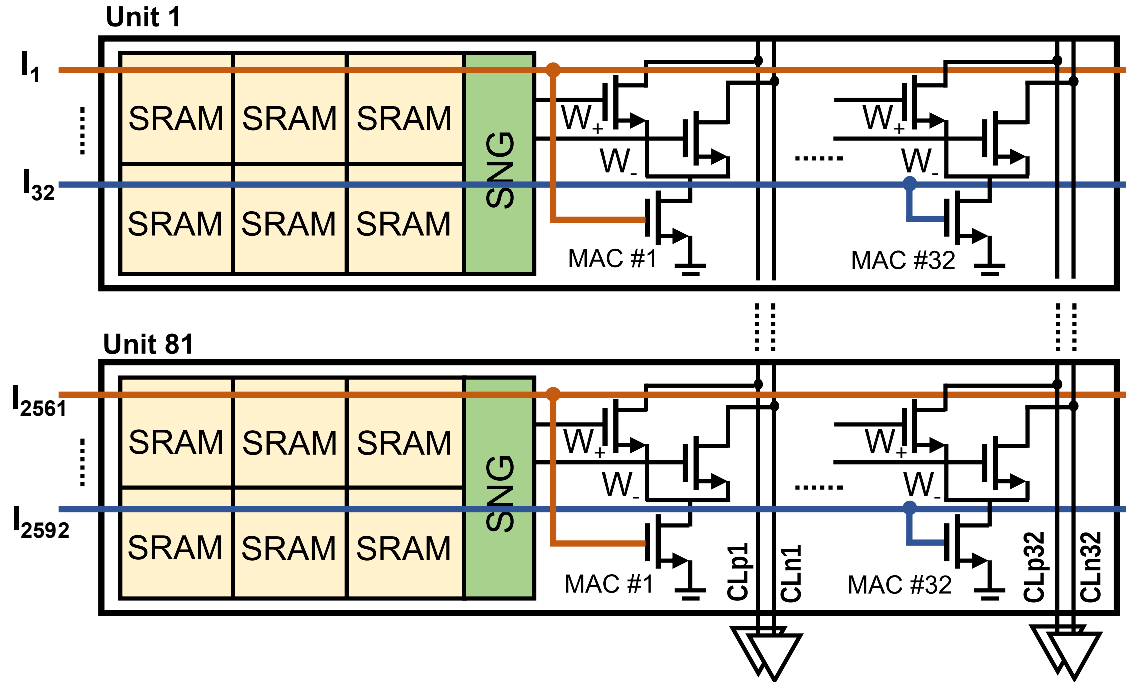


Figure 6.9: SCIM Unit with 32 MAC reuse and SCIM slice

6.3.4 Event-Based SCIM Accelerator Architecture

Figure 6.10 shows the overall architecture of SCIMITAR. Given a limited set of operations, the control logic is implemented as a finite-state machine (FSM) controlled through a set of programmable registers. The exact programmability is discussed later. The *I/O interface* transfers ROIs to input SRAMs and outputs/maxima from the output SRAM. SCIMITAR supports two modes of operation: sparse, or neuromorphic, using a single 64x64 ROI with up to eight time channels, and dense, using up to eight frame-based, grayscale 64x64 ROIs concurrently. Based on that, the architecture is organized into eight *columns*, each consisting

of an *input SRAM, staging buffer*, and a *SCIM bank*. There are eight input SRAMs, each provisioned to double buffer one time channel of a 64x64 pixel ROI. Input SRAM width is provisioned to hold 64 1-bit values, and multiple rows can store multi-bit inputs, as described below.

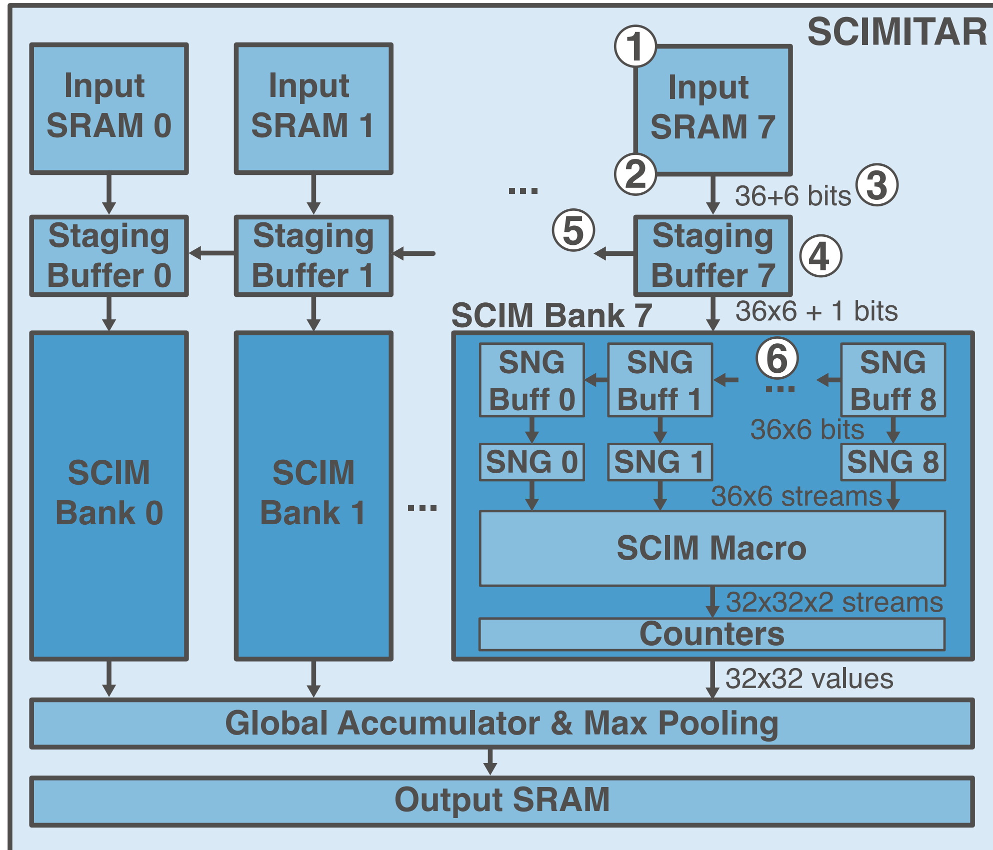


Figure 6.10: SCIMITAR architecture block diagram.

Values from input SRAM are first read into staging buffers, then optionally rotated, and passed onto the SCIM banks. Since, as described in the previous Section, the SCIM macro can only process half of the row at a time, staging buffers are provisioned for 36 6-bit values. This includes the overlap needed to avoid gaps in convolution coverage. Within each SCIM bank, the input values are written to the SNG buffers, where they are used to generate SC streams when the computation starts. Each bank has nine SNG buffers, which collectively hold nine rows of 36 values, making it possible to unroll 9x9 convolutional filters spatially.

This spatially unrolled convolutional window is similar to the one used in [5], as it maximizes spatial reuse opportunities.

The weights are pre-loaded in the SCIM, and their streams are generated in-situ, as described in the previous Section. Within each macro, a sliding 9x9 convolution is performed across nine input rows, generating 32 outputs, for 32 filters, for a total of 1024 outputs per bank. Outputs of each compute line, are fed into counters. After computation is finished, counter outputs are sent to the global accumulator block. In the sparse mode, respective outputs of up to eight counters are added implementing the combined 9x9x8 filter size. In the dense mode, outputs of banks are not accumulated, as only 9x9x1 filters are supported.

We will now describe certain architectural optimizations, indicated in Figure 6.10 using numbered circles.

① *Transposed Memory Layout*

In both frame- and ROI-based approaches, a given pixel can "fire" multiple times during the same time window, as event rates are much higher than frame rates [231]. Certain algorithms saturate, constraining pixel values to -1/0/+1, while others accumulate events, meaning that active pixels can have values proportional to their event count. SCIMITAR can work with either approach, while also supporting dense, grayscale frames with up to 6-bit precision.

To support seamless transitions between different input precisions, we use a transposed memory layout as shown in Figure 6.11. Each memory word consists of a 6-bit *next_row_id*, and 64 bits of data D. Instead of each memory word storing a set of all bits, it instead stores one bit position for a certain number of values. When using 6-bit precision, 64 values will be stored in 6 words, for 5-bit precision - 5 words, and so on. Thanks to this approach we avoid wasting memory capacity when using lower precision, as well as memory access energy, compared to a non-transposed format. Precision is programmed into one of the configuration registers of SCIMIAR, so that control logic knows how many reads are required for a given

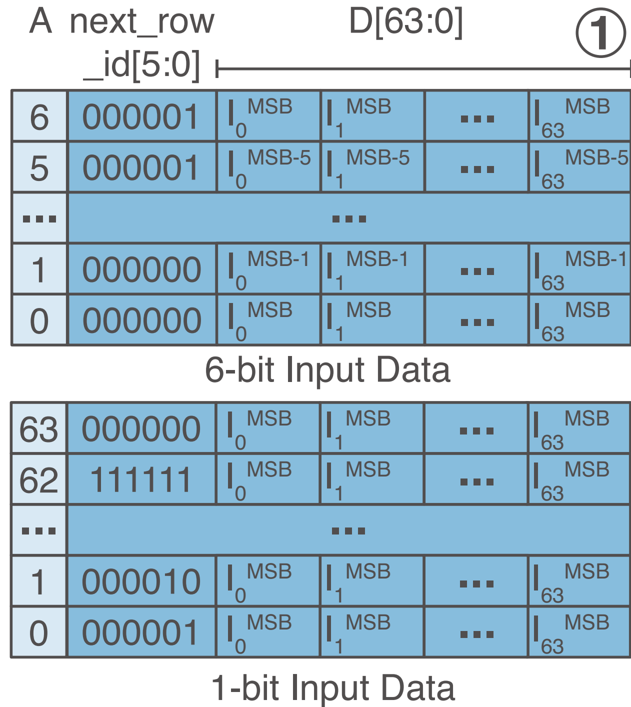


Figure 6.11: Transposed memory layout for 6-bit (top) and 1-bit (bottom) input data.

bitwidth. In conventional, fixed-point architectures, such layout might increase memory access latency, but SC stream latencies allow us to effectively hide it without performance impact. The *next_row_id* associated with each memory word is used for zero detection, described below.

② *Channel Load Skipping*

While ROI processing provides a significant reduction in memory and computation compared to full frames, individual ROIs are also highly sparse. To take advantage of this sparsity, we propose to embed additional information in input memory to avoid storing and loading parts of the ROI that contain no events. Since data in memory is organized in the form of rows, we consider two levels of granularity: *row* and *channel* skipping. The former will skip any slice of 64x8 pixels (one row across all eight time channels) if all values are zero. The latter will skip any slice of 64 pixels (one row, one time channel), if all values are

zero.

To support this functionality, input memory contains the next row index, which indicates the index of the next non-zero row stored in the subsequent address. In case of row skipping, the next row index is shared across all eight input SRAMs. For channel skipping, each SRAM has its own next row index information. To evaluate potential storage compression, we used an in-house dataset described in Section 6.4, partitioned it into 64x64x8 ROIs, and calculated the memory required for each of the ROIs, including next row index information. Results are shown in Figure 6.12. Row skipping reduces storage requirements by 2.36X on average, while channel skipping does so by 8.88X, on average, even including indexing overheads. Given those results we opt to implement channel skipping in SCIMITAR.

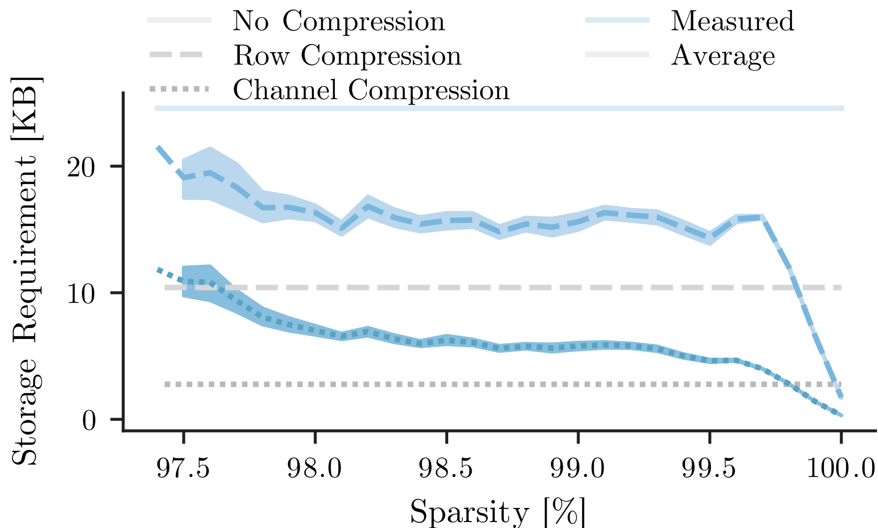


Figure 6.12: ROI memory requirements for different compression schemes.

Channel skipping is implemented in local control logic, on a column-by-column basis. Whenever reading a word from input memory, if $next_row_id$ is more than $current_row + 1$, where $current_row$ is the index of the currently read row, provided by the global control FSM, local control logic skips the next NxP reads. N is equal to the $next_row_id - current_row$, and P is equal to input precision. This is shown schematically in Figure 6.13. In other words, local input SRAM control will wait until global FSM catches up to its next non-zero

row. The first row in any given ROI is always read, as no *next_row_id* is initially known.

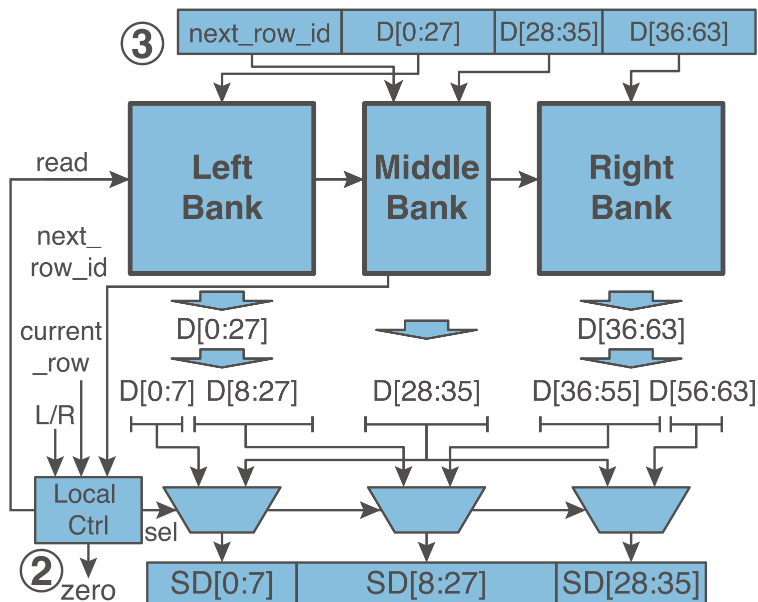


Figure 6.13: Channel load skipping and half-row multiplexing using partitioned input SRAM.

③ *Half-Row Multiplexing*

The SCIM Macro can only process half of the 64-wide row at a given time. Given the relatively high energy cost of accessing SRAM, it might seem prudent to store half-rows separately in memory, to save on accesses. However, to avoid a gap in convolution coverage, each half needs to include the same 8-pixel overlap region. Further, *next_row_id* information would need to be stored with each half-row. Instead of storing $64 + 6$ bits, we would need to store $2x(36 + 6) = 84$ bits, leading to a 20% storage overhead. Instead, we partition each input SRAM into three physical banks: left (bits 0-27), middle (bits 28-35), and right (bits 36-63), as shown in Figure 6.13. A signal from the control FSM (L/R - left/right) decides which banks are accessed (left-middle, or middle-right) and multiplexes the outputs to appropriate positions of the staged data SD. This approach avoids storage overheads, while saving access energy. *next_row_id* is stored in the middle bank, as it is always accessed.

④ *Deserializing Staging Buffers & Zero Indicator*

Input values which transposed, or bit-serialized, in memory, need to be transposed before being used to generate stochastic streams. To do this, we use deserializing staging buffers, shown in Figure 6.14. Whenever a certain bit position is read from input SRAM, global control logic indicates which bits in the buffers are enabled. Depending on the required precision, parts of the buffer can remain unused. Each staging buffer can hold up to $6x64 = 384$ bits of staged, deserialized data SDD. To further improve efficiency, staging buffers contain a zero indicator bit. Upon detecting one or more zero rows, using the $next_row_id$, local control will also set the zero indicator bit in its staging buffer, as shown in Figures 6.13 and 6.14. This bit is used downstream to gate the SNGs for that row, further saving energy.

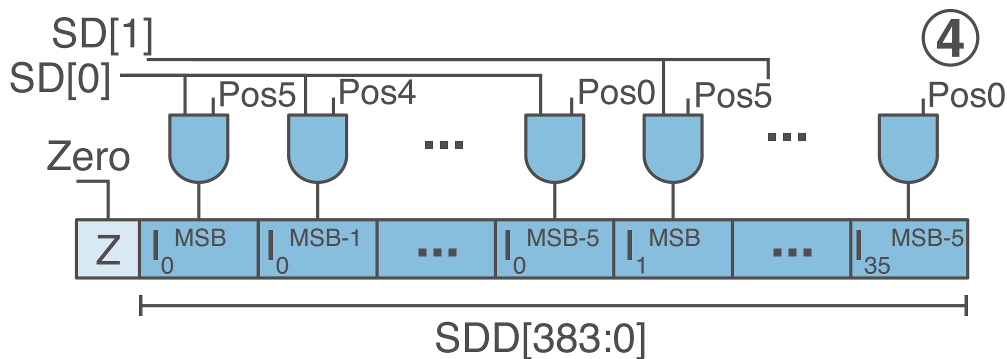


Figure 6.14: Deserializing staging buffers with zero bit indicator.

⑤ *Time Channel Overlap*

SCIMITAR supports up to 8 time channels in each ROI. In some applications subsequent "frames" can be completely non-overlapping, meaning their time channels cover non-overlapping windows. For example, if using 1ms time-channels, the first reconstructed frame covers the first 8ms, the second one the next 8ms and so on. However, temporal resolution can be vastly improved if there is an overlap between subsequent "frames". For example, each subsequent reconstruction could be shifted by 1ms, where the reamining 7ms overlap. Given that in SCIMITAR, filter time channels are assigned to physical SCIM macros, each of which is connected to its own input SRAM, naively supporting such overlap would require reloading the entire ROI, as time channels would need to be physically moved between in-

put banks. To provide seamless support for overlapping time-channel ROIs, we propose to connect staging buffers in a form of a circular buffer, as shown in Figure 6.15.

Initially, time channels are properly aligned to columns, for the first 8 time steps. Half-rows can be loaded directly into staging buffers and passed to their respective SCIM banks. After processing, time channel $t = 0$ is replaced with time channel $t = 8$ in column 0. After loading each row to the staging buffers, they are rotated once, so that channel $t = 8$ ends in column 7, channel $t = 7$ in column 6, and so on. Using this approach, time channels can be overlapped with minimum number of memory accesses. The latency of rotating staging buffers is again hidden using stochastic stream processing latencies. In the worst case scenario, SCIMITAR needs to hide 6 cycles of memory reads (for 6-bit precision) and 7 cycles of rotation, for a total of 13 cycles. This is much lower than typical stream lengths used, however, it might not be the case when early termination is used, as described in the next Section.

⑥ *Sliding Convolution Window*

Within each SCIM bank, there are 9 SNG buffers, organized as shift registers. This is done to emulate the vertically sliding convolution window. Whenever a new row is loaded, all previous ones are shifted, with the last one ("highest") shifted out. Since filter weights have a fixed position in memory with respect to the values in SNG buffers, this is equivalent to shifting the 9x9 convolutional window in each bank one row down. This approach is similar to the one used in [5]. The zero bit is propagated with the values in SNG buffers, and is used to get stream generation when all values are zero to save energy.

Figure 6.16 shows how computational energy efficiency if affected by the above optimizations. The use of transposed layout, row multiplexing, and deserializing buffers lowers the energy by 1.64X, when using low precision data. Adding channel skipping on top of it, further reduces energy by 1.28X. Finally, time channel overlap and sliding convolutional window allow us to save additional 14% on top of the above, for a 2.8X overall improvement.

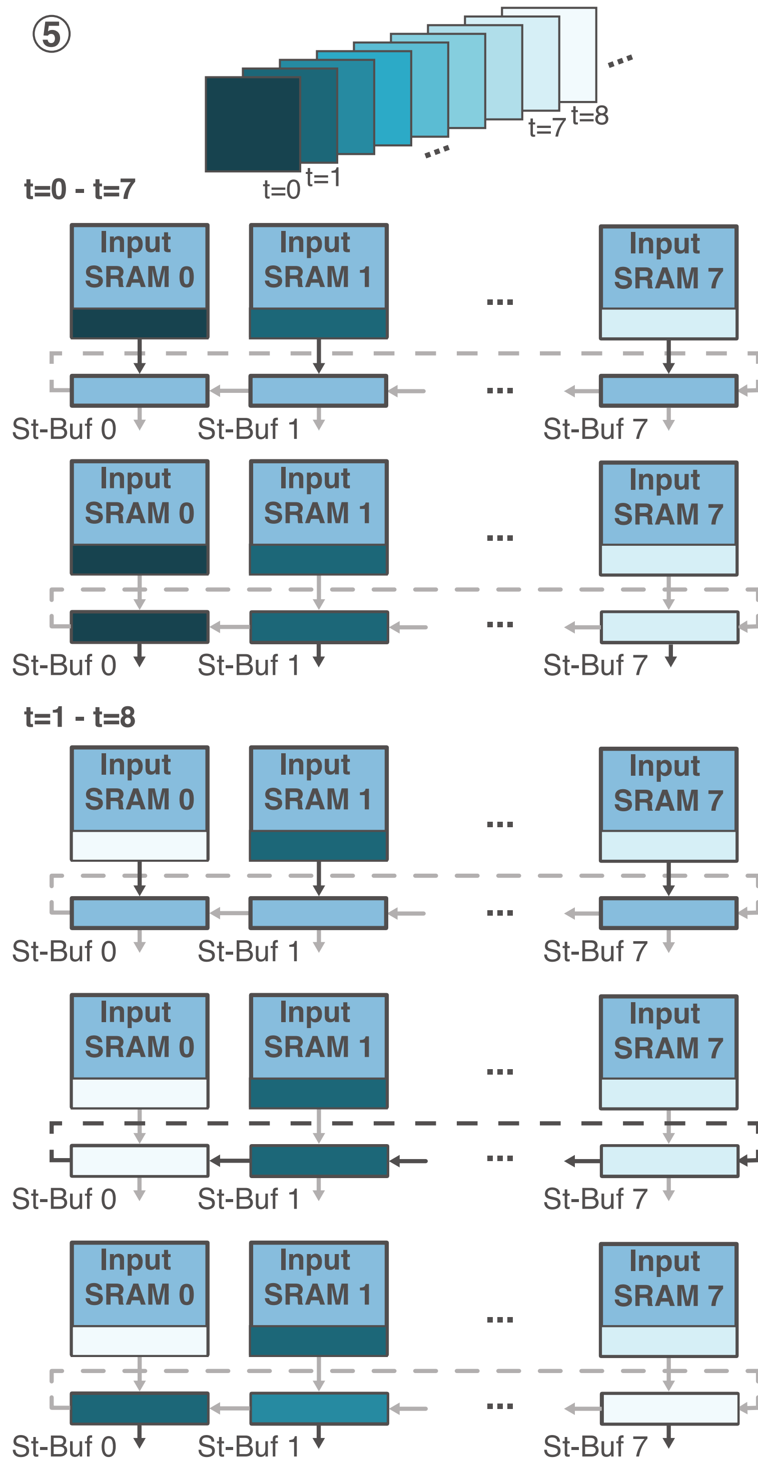


Figure 6.15: Time channel overlap support using circular staging buffers.

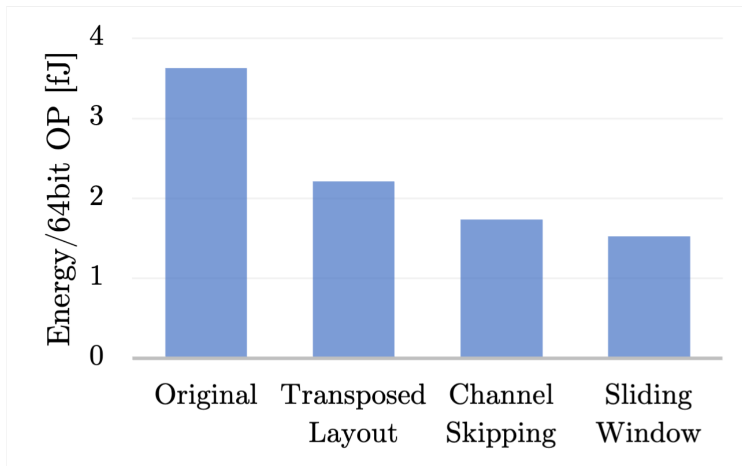


Figure 6.16: Impact of proposed optimizations on computational energy efficiency of the SCIMITAR architecture. Efficiency calculated on 99% sparse input data assuming 64-bit long SC streams.

6.3.5 Multi-Level Early Termination

Stochastic computing has a known property of progressive precision, where the output counter value converges onto the final result with each bit of the stream being processed [64]. Hence, at every cycle of computation, we can treat the counter output as an estimate of the final result. For example, counting the 1s resulting from a 16-cycle computation might result in a count of 8. But if this is the case, we expect that if we only run part of the computation the ratio should still be about 50% 1s. This leads to the concept of early termination (ET) [64]. Since early partial results from running a computation approximate the final result, we can judge whether or not we are likely to care about the result of a computation before the computation is complete.

The tradeoff with using results from shorter streams is that the shorter the stream is, the more likely it is to have a large error. For example, if we expect a result of 50% for a long stream, we might still expect about 50% for a short stream. But an error of 1 on a 16-bit stream could be 9/16 whereas an off-by-1 error for a 4-bit stream could be 3/4 which

is significantly more serious. To avoid this size of the error, we do not use early termination on the first 8 bits of the stream and only turn it on after that.

In our application, we are looking for peaks, or maxima, of the filter convolution. This means that if a value is negative or even just very close to 0 after some number of cycles, it will most probably not be a peak. Although the sum of the first few bits of a computation may have a significant error, we can make a very confident prediction about whether or not a result will be a peak well before the computation is finished. This is the concept of early termination. Using early termination, we can save power on many different inputs and latency when processing inputs that do not contain objects and thus do not produce any peaks above the object identification threshold.

In our early termination implementation, we periodically check the result of the stream to see if it is below a threshold that would allow us to discard the pixel as a peak candidate. In an ideal case for early termination, we could check the count against a threshold every cycle, but in our case, we have unrolled positive and negative streams in time (alternating every 8 bits) so the value is only fair to compare to early termination every 16 cycles when we have computed an equal number of positive and negative streams. This gives us the potential to reduce the time of computations by up to 75%, assuming 64-bit streams, based on a threshold that can be chosen for the application, with a high degree of confidence that we will not lose any peaks that we care about.

We apply early termination to three levels of computation, as shown in Figure 6.17:

- Level 1: each output counter contains a comparator, that compares the value against a global threshold. If the value is below the threshold at the time of the comparison, we disable the counter until the next computation starts. Level 1 early termination can therefore save output counter energy.
- Level 2: if all output counters corresponding to the same Gabor filter are disabled, we then disable the stream generation for weights of that filter. This further reduces

computation energy.

- Level 3: if all output counters are disabled, we stop computation altogether, and move to the next iteration. This not only saves energy but also latency, as it effectively translates to shorter stream lengths.

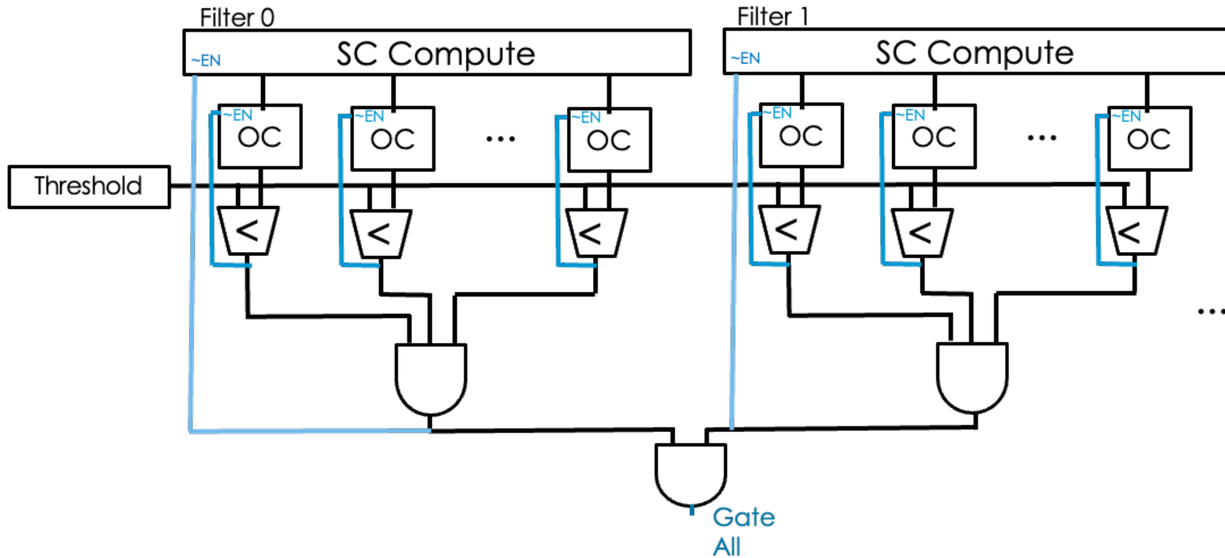


Figure 6.17: Schematic implementation of 3-level early termination.

We have evaluated early termination on a scaled-down, digital implementation of the SCIMITAR architecture, implemented on an FPGA. We tested it on selected ROIs containing objects from two in-house datasets containing various flying objects. Results, for varying stream lengths, are shown in Table 6.2. From processing using 64-bit long streams, level 3 of early termination alone can provide up to 2.6X speedup.

Table 6.2: Average ROI processing latency in cycles, with and without early termination for different stream leangths.

Stream	ROI Latency w/o ET [cycles]	ROI Latency w/ ET [cycles]	ET Speedup
16	2552	1943	1.3
32	3576	2010	1.8
64	5624	2131	2.6

6.4 Evaluation

6.4.1 Accuracy

While algorithmic performance is beyond the scope of this work, we evaluate how the use of approximate computation, compares with ”exact” methods, i.e. using floating-point computation. For comparison, we use spatio-temporal, 9x9 Gabor filters with eight time channels, each consisting of 1 ms worth of accumulated events. We convolve those filters with frames generated using our in-house datasets of flying objects. The datasets are highly sparse (96-98% sparsity). Frames are divided into 64x64 ROIs. For ”exact” output, we calculate the results using Matlab. SCIMITAR results are based on a digital implementation, which matches the behavior of the SCIM, written using Verilog RTL and executed on an Alveo U200 FPGA accelerator card. We use three different stochastic stream lengths - 16, 32, and 64 bits, all with early termination. For the 64-bit scenario, we also evaluate the accuracy without ET. We consider 3 accuracy metrics: ROI agreement, filter agreement, and distance error. ROI agreement is the percentage of filter maxima that were found in the same ROI in both exact and approximate computation. Within those, filter agreement is the percentage of matching peak-filter pairs between exact and approximate evaluation, and distance error is the average absolute distance, in pixels, between exact and approximate peaks.

Results of our evaluation are shown in Table 6.3. We see that approximate computation

can match the peaks of between 80 and 90% for the ROIs considered. Within matched ROIs, there is a very high correlation between exact and approximate peaks (> 99%), and the peaks found using SC computation are generally within a few pixels of their "exact" locations. It is important to note, that the use of SC makes it possible to trade off performance and accuracy - if higher fidelity is needed, longer stream lengths can be used. Further, early termination can be disabled to further reduce the error in computation. Finally, the results in Table 6.3 do not incorporate any of the techniques developed in Chapter 4, which can significantly improve the precision of SC computation. A more detailed evaluation of the SCIMITAR architecture on an end-to-end tracking application is a subject for future work.

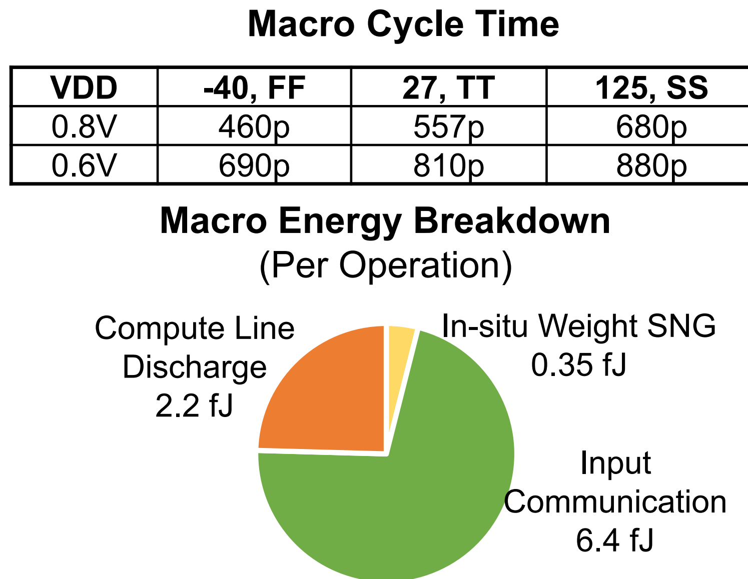
Table 6.3: Accuracy metrics of approximate computation in tracking applications using Gabor filters.

Stream	ET	ROI Agreement [%]	Max Filter Agreement [%]	Distance Error [pixels]
64	No	88	99	3
64	Yes	84	99	4.3
32	Yes	78	99	4.1
16	Yes	79	99	3.9

6.4.2 Hardware Evaluation

The SCIM macro is designed and simulated in 12nm technology using the extracted netlist after layout, using the Cadence Virtuoso tool. The macro includes a digital controller, input buffers, SCIM cell arrays, and read/write circuitries. The SRAM cell is custom-designed, instead of using the foundry's cell to pass the standard logic rules. A replica reference column is used to track the process, voltage, and temperature (PVT) variation on the chip and controls how long the compute line is discharged. The timing constraint of the SCIM macro is simulated over multiple temperature, process, and voltage corners shown in Figure

6.18. The worst-case cycle time is 880p sec for SS corner, 125 Celsius, and 0.6V supply, which meets the target system frequency requirement of 1GHz. The macro energy is measured by running a transient simulation using the extracted netlist and integrating the supply power waveforms. A dense load is used for inputs and weight, which means they are nonzero and have a 0.5 probability of being 1. Figure 6.18 shows the breakdown of the macro energy. The energy is normalized by the number of 6-bit operations. A MAC is counted as two operations and the 6-bit operation is equivalent to processing 64 SC bits. Input communication energy is the biggest component of energy consumption, followed by compute line discharge, and in-situ weight SNG. The total energy consumption per operation is 8.95 fJ, which is equivalent to 112 TOPS/W.



Macro Cycle Time

VDD	-40, FF	27, TT	125, SS
0.8V	460p	557p	680p
0.6V	690p	810p	880p

Figure 6.18: SCIM Macro timing (top), and energy breakdown (bottom).

For the system-level evaluation, we have implemented buffers and SNGs in Verilog RTL and synthesized them using the same 12nm technology. SRAM energy is modelled based on 12nm SRAM compiler tools. We estimate energy consumption assuming pessimistic 50% switching probability of the stochastic streams, and three levels of input sparsity: dense (0% sparse), 90%, and 99%. Further, we incorporate the impact of the optimizations proposed

in Section 6.3.4 by statically profiling the datasets discussed before, and adjusting memory and SNG energy.

Overall results, with a detailed breakdown, are shown in Figure 6.19. On a dense load, SCIMITAR consumes 12.9 fJ per 64-bit SC operation, translating to energy efficiency of 77.2 TOPS/W. At 90% sparsity, thanks to proposed optimizations and the implicit gating of analog SCIM logic with zero inputs, per-operation energy is only 1.89 fJ, or 527 TOPS/W. At 99% sparsity, the energy is only 1.46 fJ/OP, translating to efficiency of 680 TOPS/W. SCIMITAR offers very high energy-efficiency for processing ROI-based event-data, that can scale with sparsity thanks to analog circuit computation, and additional proposed optimizations. A comparable in-memory accelerator demonstrated in [268] achieves 60 TOPS/W with 6-bit precision, in a comparable 16nm technology.

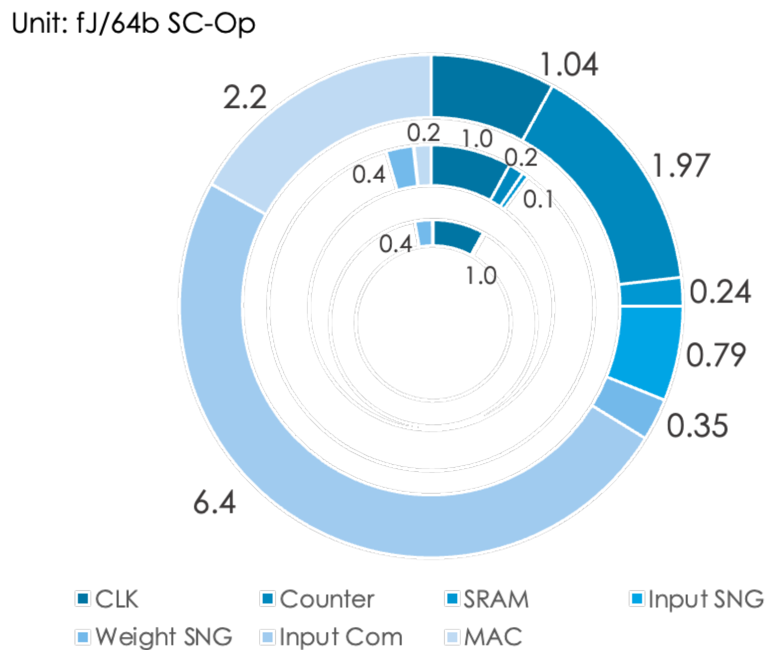


Figure 6.19: Energy breakdown of the SCIMITAR components for dense (outer circle), 90% sparse (middle circle), and 99% sparse (inner circle) workloads. Energy is calculated using 64-long stochastic streams.

6.5 Conclusion

In this Chapter, we proposed SCIMITAR: Stochastic Computing In-Memory In-situ Tracking ARchitecture for Event-Based Cameras, an accelerator for high-speed object tracking. SCIMITAR uses stochastic compute-in-memory (SCIM) for highly efficient analog processing, in-situ stream generation for compact implementation, and a host of optimizations for utilizing input sparsity. SCIMITAR provides unparalleled throughput for ROI-based processing, with both latency and energy that can scale with sparsity. We demonstrate SCIMITAR energy-efficiency using detailed circuit level simulations.

CHAPTER 7

Conclusion

This chapter reviews the key contributions of this dissertation and outlines directions for future work.

7.1 Overview of Contributions

In this dissertation a series of techniques have been proposed to enable computationally and memory-heavy machine learning models to be implemented on resource-constrained edge devices. These techniques are applicable in both software (3PXNet), including open-sourced libraries, as well as hardware (ACOUSTIC, GEO, SASCHA, SCIMITAR), including FPGA and ASIC implementations. They span the use and combinations of binarization (3PXNet), sparsity (3PXNet, SASCHA, SCIMITAR, stochastic computing (ACOUSTIC, GEO, SASCHA, SCIMITAR) and in-memory compute (SCIMITAR).

7.1.1 3PXNet

3PXNet, described in Chapter 2, demonstrates the first combination of binarization and sparsity in a software implementation. The 3PXNet approach relies on overcoming the challenges of combining both techniques without losing their benefits. It is done through a process of Pruning, Permuting, and Packing trained network weights. 3PXNet opens up new possibilities for embedded systems developers. 3PXNet results have been demonstrated on a host of microcontroller devices. We have shown up to 300X and 38X model size reduction

compared with 8-bit fixed-point and dense binarized models and up to 25X improvement in runtime and energy. In practice, this enables the use of models that previously could not be deployed on a particular platform due to memory or runtime constraints. Alternatively, it can lower the system cost by enabling the use of cheaper parts with lower performance or memory capacity. 3PXNet is based on commonly used frameworks: Pytorch for training and the C programming language for deployment, making it broadly and easily applicable. Further, a compiler makes translation from training to deployment seamless and user-friendly.

7.1.2 ACOUSTIC & GEO

Both ACOUSTIC and GEO, presented in Chapters 3 and 4 are system-level stochastic computing accelerator architectures targeted at convolutional neural networks. ACOUSTIC is the first comprehensive edge design that not only uses stochastic computing for acceleration but also considers the programming model, scheduling, and memory hierarchy. Previous SC works have focused only on the compute part, which made a holistic analysis impossible. By putting SC in the context of an entire system, ACOUSTIC develops a set of crucial insights regarding the most important benefits of using SC and how to best use them for system-level benefits. ACOUSTIC then provides an baseline for SC accelerators, as well as an important first step into their broader adoption. ACOUSTIC results have been demonstrated using synthesis, performance simulation, FPGA, and even custom ASIC in Global Foundries 14nm technology. The ACOUSTIC architecture delivers server-class parallelism within a mobile area and power budget - a 12mm2 accelerator can be as much as 38.7x more energy-efficient and 72.5x faster than conventional fixed-point accelerators. It can also be up to 79.6x more energy-efficient than state-of-the-art stochastic accelerators and can be implemented in order of magnitude less area than recent SC-based accelerators, delivering real-time performance in a mobile/IoT energy/area envelope. The availability of working hardware devices justifies performance claims and makes it possible for parties interested in using SC to evaluate its viability.

GEO builds up on the ACOUSTIC design by fixing its biggest shortcomings: accuracy loss, reliance on memory bandwidth, and inefficient stream conversion. GEO does this through a set of complementary techniques: trained, shared stream generation, progressive shadow buffers, partial binary accumulation, and near-memory computation. Thanks to these optimizations, GEO improves accuracy by 2.2-4.0% points compared to other SC-based accelerators while also being 4.4X faster and 5.6X more energy efficient. GEO can compete with fixed-point implementations with similar accuracy and area while delivering up to 5.6X throughput and 2.6X energy-efficiency gains. GEO results have been verified in synthesis and using performance simulation. By closing the accuracy gap between SC and fixed-point computation, without sacrificing performance, GEO marks another significant step towards the broader adoption of SC-based accelerators.

7.1.3 SASCHA

SASCHA provides the important next step in the evolution of SC-based accelerators. Not only does it enable exploiting sparsity in weights to improve performance, but it is also more flexible than ACOUSTIC and GEO, due to its GEMM-based nature. SASCHA PE design, scheduler, and architecture are designed to overcome non-trivial complications in combining SC and sparsity. This goal was achieved through an extensive processing element design exploration factoring in different aspects of SC, building a scheduling algorithm to improve utilization, and a novel use of SC bit-slicing to extract sparsity from even dense weights. SASCHA results have been demonstrated using synthesis results, providing a strong foundation for our claims. The ability to process general matrix multiplications makes it possible for SASCHA to run models beyond just CNNs, like multi-layer perceptrons (MLPs) or transformers. Further, SASCHA does not require users to sacrifice dense network throughput, thanks to the ingenious use of parallel stream processing in SC, making it more versatile than many sparse fixed-point accelerators.

7.1.4 SCIMITAR

SCIMITAR is an apt conclusion to this dissertation, as it combines most of the themes discussed in other parts: stochastic computing, sparsity, compute-in-memory, low-precision, and domain-specific acceleration. SCIMITAR is specifically designed towards for the emerging event-based cameras, and takes advantage of their sparse, low-latency data streams to accelerate object detection and tracking applications. Not only does SCIMITAR use state-of-the-art stochastic compute-in-memory (SCIM) to provide very high energy-efficiency even on dense load, it also incorporates circuit and architectural optimizations that allow us to scale performance with input sparsity. On top of that, the introduction of the in-situ SNG technique can significantly cut down the area requirements of SCIM designs, making them more suitable for edge applications. Despite using inherently approximate computation, SCIMITAR provides results closely matching "ideal", floating-point computation, and techniques are available for improving its accuracy further.

7.2 Directions for Future Work

Potential research avenues to expand on the work presented in this dissertation are discussed below.

7.2.1 Exploration of Stochastic Computing Accelerators

While the various architectures presented here represent, or represented at the time of their publication, the state-of-the-art in ML acceleration, they were largely designed by hand. One way of improving them would be through automated design-space exploration, something that has been extensively applied to conventional accelerators, for example in [129]. As we have shown in Chapter 5, SC-based accelerators have different optimization axes, not present in fixed-point architectures, that can lead to a different set of optimal parameters.

Because of that, automated design space exploration that takes into account the peculiarities of stochastic computing could uncover new and interesting insights into how to best design such devices. Further, the evaluation in this thesis is largely limited to convolutional vision models, whereas the ML landscape is moving towards transformer-based models. This limitation was caused by the slow training time of SC network models, something that is beyond the scope of this dissertation. However, given the recent advancements in training, SC-based transformer, and other models will soon become a possibility. Designing architecture for them will become a new challenge. While SASCHA presents a good starting point, being a GEMM-based accelerator, it remains to be seen how effective it would be at running transformer models. Finally, if SC accelerators become broadly used, they will require robust compiler support. While a compiler for ACOUSTIC and GEO has been developed, it can only support a limited subset of operations in basic models and is unable to find and exploit code optimization opportunities. Much more work is required on this front.

7.2.2 Analog Stochastic Computing

While SCIMITAR, presented in Chapter 6, successfully marries stochastic computing and analog, in-memory computation, there are plenty of potential extensions for this work. First, we found new range-extending techniques, which can maintain accuracy at lower SC stream lengths, significantly improving performance. They were initially elaborated in a purely digital setting, currently in publication, but can be applied to analog, in-memory stochastic computation as well. Second, the combination of analog computation, and SC robustness remain to be explored. We know that, due to its non-positional representation, stochastic computing is very timing error-tolerant [201]. Since analog circuits often burn a lot of area and power for margining and tolerance, SC robustness could be used to design more streamlined circuitry. Various other techniques could also be used: DVFS, sub-threshold operation, pipelining, etc. Ideally, all of those components could be combined in an automatic design exploration tool, as described in the previous subsection.

7.2.3 Extending 3PXNet

Despite impressive performance gains, 3PXNet suffers from accuracy losses. Some gains have been made in recent years in improving training and network architecture to recover some precision, but the amount of information stored in a single bit remains a fundamental limitation. To alleviate that, multi-bit binarized networks have been proposed [269], which can use arbitrary precision for each of the operands, while still using efficient "SIMD" XNOR multiplication. To maintain performance gains over optimized, 8-bit libraries, precision is limited to two or three bits, but even that can drastically improve accuracy [269]. Unfortunately, existing work relies on using the so-called "unipolar" multiplication, with inputs being 0 or 1, which is not as efficient as bipolar, XNOR multiplication. Our ongoing efforts have shown that XNOR multiplication can be used with a careful choice of activation functions. Further, there is ongoing work on combining multi-bit XNOR networks with 3PXNet, to deliver accuracy-competitive networks with minimal memory and runtime.

REFERENCES

[1] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," *arXiv preprint arXiv:1409.1556*, 2014. arXiv: 1409.1556 ISBN: 0950-5849.

[2] Y. H. Chen, T. Krishna, J. Emer, and V. Sze, "Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks," *IEEE journal of solid-state circuits*, vol. 52, no. 1, pp. 127–138, 2016. arXiv: 1512.04295 ISBN: 978-1-4673-9466-6.

[3] T. Jia, Y. Ju, and J. Gu, "A Dynamic Timing Enhanced DNN Accelerator With Compute-Adaptive Elastic Clock Chain Technique," *IEEE Journal of Solid-State Circuits*, vol. 56, pp. 55–65, Jan. 2021.

[4] A. Sayal, S. Fathima, S. T. Nibhanupudi, and J. P. Kulkarni, "COMPAC: Compressed Time-Domain, Pooling-Aware Convolution CNN Engine With Reduced Data Movement for Energy-Efficient AI Computing," *IEEE Journal of Solid-State Circuits*, vol. 56, pp. 2205–2220, July 2021.

[5] W. Romaszkan, T. Li, T. Melton, S. Pamarti, and P. Gupta, "ACOUSTIC : Accelerating Convolutional Neural Networks through Or-Unipolar Skipped Stochastic Computing," in *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 768–773, 2020.

[6] A. Sayal, S. Fathima, S. S. T. Nibhanupudi, and J. P. Kulkarni, "All-Digital Time-Domain CNN Engine Using Bidirectional Memory Delay Lines for Energy-Efficient Edge Computing," in *2019 IEEE International Solid- State Circuits Conference - (ISSCC)*, vol. 49, pp. 228–230, IEEE, 2019. Issue: 4.

[7] A. Biswas and A. P. Chandrakasan, "Conv-RAM: An energy-efficient SRAM with embedded convolution computation for low-power CNN-based machine learning applications," in *2018 IEEE International Solid - State Circuits Conference - (ISSCC)*, (San Francisco, CA), pp. 488–490, IEEE, Feb. 2018.

[8] W. Romaszkan, T. Li, and P. Gupta, "3PXNet: Pruned-Permuted-Packed XNOR Networks for Edge Machine Learning," *ACM Transactions on Embedded Computing Systems*, vol. 19, no. 1, pp. 1–23, 2020.

[9] W. Romaszkan, T. Li, R. Garg, J. Yang, S. Pamarti, and P. Gupta, "A 4.4–75-TOPS/W 14-nm Programmable, Performance- and Precision-Tunable All-Digital Stochastic Computing Neural Network Inference Accelerator," *IEEE Solid-State Circuits Letters*, vol. 5, pp. 206–209, 2022.

[10] T. Li, W. Romaszkan, S. Pamarti, and P. Gupta, "GEO : Generation and Execution Optimized Stochastic Computing Accelerator for Neural Networks," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1–6, 2021.

[11] W. Romaszkan, T. Li, and P. Gupta, "SASCHA—Sparsity-Aware Stochastic Computing Hardware Architecture for Neural Network Acceleration," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, pp. 4169–4180, Nov. 2022.

[12] M. Horowitz, "Computing's Energy Problem (And What We Can Do About It)," in *Digest of Technical Papers - IEEE International Solid-State Circuits Conference*, vol. 57, pp. 10–14, 2014. ISSN: 01936530.

[13] Y. H. Chen, J. Emer, and V. Sze, "Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pp. 367–379, 2016. ISSN: 02721732.

[14] N. P. Jouppi, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, C. Young, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, N. Patil, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Patterson, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, G. Agrawal, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, R. Bajwa, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, S. Bates, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, D. H. Yoon, S. Bhatia, and N. Boden, "In-Datacenter Performance Analysis of a Tensor Processing Unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pp. 1–12, 2017. arXiv: 1704.04760 ISSN: 10636897.

[15] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam, "DaDianNao: A Machine-Learning Supercomputer," in *47th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 609–622, IEEE, 2014. arXiv: 1512.04295v2 ISSN: 10724451.

[16] C. Banbury, V. J. Reddi, P. Torelli, J. Holleman, N. Jeffries, C. Kiraly, P. Montino, D. Kanter, S. Ahmed, D. Pau, U. Thakker, A. Torrini, P. Warden, J. Cordaro, G. Di Guglielmo, J. Duarte, S. Gibellini, V. Parekh, H. Tran, N. Tran, N. Wenxu, and X. Xuesong, "MLPerf Tiny Benchmark," 2021. arXiv: 2106.07597.

[17] J.-s. Park, J.-w. Jang, H. Lee, D. Lee, S. Lee, H. Jung, S. Lee, S. Kwon, K. Jeong, J.-h. Song, S. Lim, and I. Kang, "A 6K-MAC Feature-Map-Sparsity-Aware Neural

Processing Unit in 5nm Flagship Mobile SoC," in *2021 IEEE International Solid-State Circuits Conference-(ISSCC)*, pp. 152–153, 2021.

[18] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks," in *European Conference on Computer Vision*, (Amsterdam, The Netherlands), pp. 525–542, XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks, 2016. arXiv: 1603.05279 ISSN: 0302-9743.

[19] P. Judd, J. Albericio, and A. Moshovos, "Stripes: Bit-Serial Deep Neural Network Computing," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–12, 2016. arXiv: 1011.1669v3 ISSN: 15566056.

[20] T. W. Chin, P. I. Chuang, V. Chandra, and D. Marculescu, "One Weight Bitwidth to Rule Them All," *European Conference on Computer Vision Workshops*, 2020. arXiv: 2008.09916.

[21] J. Yu, A. Lukefahr, D. Palframan, G. Dasika, R. Das, and S. Mahlke, "Scalpel: Customizing DNN Pruning to the Underlying Hardware Parallelism," *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pp. 548–560, 2017. ISBN: 978-1-4503-4892-8.

[22] Y. L. Cun, J. S. Denker, and S. a. Solla, "Optimal Brain Damage," in *Advances in Neural Information Processing Systems*, vol. 2, pp. 598–605, 1990. arXiv: 1011.1669v3 Issue: 1 ISSN: 1098-6596.

[23] J. Faraone, N. Fraser, G. Gambardella, M. Blott, and P. H. Leong, "Compressing Low Precision Deep Neural Networks Using Sparsity-Induced Regularization in Ternary Networks," in *International Conference on Neural Information Processing*, pp. 393–404, 2017. arXiv: 1709.06262 ISSN: 16113349.

[24] S. Li, W. Romaszkan, A. Graening, and P. Gupta, "SWIS - Shared Weight bIt Sparsity for Efficient Neural Network Acceleration," in *Proceedings of 2021 TinyML Research Symposium*, vol. 1, pp. 1–8, 2021. arXiv: 2103.01308 Issue: 1.

[25] S. Yu, H. Jiang, S. Huang, X. Peng, and A. Lu, "Compute-in-Memory Chips for Deep Learning: Recent Trends and Prospects," *IEEE Circuits and Systems Magazine*, vol. 21, pp. 31–56, July 2021. Publisher: Institute of Electrical and Electronics Engineers Inc.

[26] P. A. Merolla, J. V. Arthur, R. Alvarez-Icaza, A. S. Cassidy, J. Sawada, F. Akopyan, B. L. Jackson, N. Imam, C. Guo, Y. Nakamura, and Others, "A Million Spiking-Neuron Integrated Circuit with a Scalable Communication Network and Interface," *Science*, vol. 345, no. 6197, pp. 668–673, 2014.

[27] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," in *Communications of the ACM*, pp. 1106–1114, 2012. arXiv: 1102.0183 ISSN: 10495258.

[28] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," Jan. 2015. Number: arXiv:1409.0575 arXiv:1409.0575 [cs].

[29] A. Canziani, A. Paszke, and E. Culurciello, "An Analysis of Deep Neural Network Models for Practical Applications," *arXiv preprint arXiv:1605.07678*, 2016. arXiv: 1605.07678 ISBN: 2857825749.

[30] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778, 2016. arXiv: 1512.03385 ISSN: 15737721.

[31] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going Deeper with Convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1–9, 2015.

[32] Y. G. Kim and C. J. Wu, "Autoscale: Energy efficiency optimization for stochastic edge inference using reinforcement learning," *Proceedings of the Annual International Symposium on Microarchitecture, MICRO*, vol. 2020-Octob, pp. 1082–1096, 2020. ISBN: 9781728173832.

[33] S. Wang, A. Pathania, and T. Mitra, "Neural Network Inference on Mobile SoCs," *IEEE Design & Test*, vol. 37, no. 5, pp. 50–57, 2020. Publisher: IEEE.

[34] C. J. Wu, D. Brooks, K. Chen, D. Chen, S. Choudhury, M. Dukhan, K. Hazelwood, E. Isaac, Y. Jia, B. Jia, T. Leyvand, H. Lu, Y. Lu, L. Qiao, B. Reagen, J. Spisak, F. Sun, A. Tulloch, P. Vajda, X. Wang, Y. Wang, B. Wasti, Y. Wu, R. Xian, S. Yoo, and P. Zhang, "Machine learning at facebook: Understanding inference at the edge," *Proceedings - 25th IEEE International Symposium on High Performance Computer Architecture, HPCA 2019*, pp. 331–344, 2019. Publisher: IEEE ISBN: 9781728114446.

[35] Y. Cheng, D. Wang, P. Zhou, and T. Zhang, "A Survey of Model Compression and Acceleration for Deep Neural Networks," *arXiv preprint arXiv:1710.09282*, 2017. arXiv: 1710.09282 ISBN: 9781467380263.

[36] M. Thoma, "Analysis and Optimization of Convolutional Neural Network Architectures," *arXiv preprint arXiv:1707.09725*, 2017. arXiv: 1707.09725.

[37] V. Vanhoucke, A. Senior, and M. Mao, "Improving the Speed of Neural Networks on CPUs," in *in Deep Learning and Unsupervised Feature Learning Workshop, NIPS*, 2011. ISSN: 9781450329569.

[38] V. K. Chippa, S. T. Chakradhar, K. Roy, and A. Raghunathan, "Analysis and Characterization of Inherent Application Resilience for Approximate Computing," in *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1–9, 2013.

[39] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep Learning with Limited Numerical Precision," in *International Conference on Machine Learning*, pp. 1737–1746, 2015. arXiv: 1502.02551 ISSN: 19410093.

[40] S. Zhou, Y. Wu, Z. Ni, X. Zhou, H. Wen, and Y. Zou, "DoReFa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients," 2016. arXiv: 1606.06160.

[41] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mane, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viegas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems," Mar. 2016. Number: arXiv:1603.04467 arXiv:1603.04467 [cs].

[42] Z. Yao, Z. Dong, Z. Zheng, A. Gholami, J. Yu, E. Tan, L. Wang, Q. Huang, Y. Wang, M. W. Mahoney, and K. Keutzer, "HAWQV3: Dyadic Neural Network Quantization," *arXiv preprint arXiv:2011.10680*, pp. 1–15, 2020. arXiv: 2011.10680.

[43] Z. Cheng, D. Soudry, Z. Mao, and Z. Lan, "Training Binary Multilayer Neural Networks for Image Classification using Expectation Backpropagation," *arXiv preprint arXiv:1503.03562*, 2015. arXiv: 1503.03562 ISBN: 9781479909209.

[44] M. Courbariaux, Y. Bengio, and J.-P. David, "BinaryConnect: Training Deep Neural Networks with Binary Weights During Propagations," *Advances in Neural Information Processing Systems*, vol. 28, pp. 3123–3131, 2015. arXiv: 1511.00363 ISSN: 10495258.

[45] M. Kim and P. Smaragdis, "Bitwise Neural Networks," *arXiv preprint arXiv:1601.06071*, vol. 37, 2016. arXiv: 1601.06071 ISBN: 9781538646588.

[46] I. Hubara, D. Soudry, and R. E. Yaniv, "Binarized Neural Networks," *Advances in neural information processing systems*, vol. 29, no. Nips, pp. 1–9, 2016. arXiv: 1602.02505.

[47] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers, "FINN : A Framework for Fast , Scalable Binarized Neural Network Inference," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 65–74, 2017.

[48] N. J. Fraser, Y. Umuroglu, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers, “Scaling Binarized Neural Networks on Reconfigurable Logic,” in *Proceedings of the 8th Workshop and 6th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures and Design Tools and Architectures for Multicore Embedded Computing Platforms*, (Stockholm, Sweden), pp. 25–30, Association for Computing Machinery, 2017. arXiv: 1701.03400 ISBN: 9781450348775.

[49] Y. Li, Z. Liu, K. Xu, H. Yu, and F. Ren, “A 7.663-TOPS 8.2-W Energy-efficient FPGA Accelerator for Binary Convolutional Neural Networks,” in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays - FPGA '17*, (Monterey, California, USA), pp. 290–291, Association for Computing Machinery, 2017. arXiv: 1702.06392 Issue: March.

[50] R. Zhao, W. Song, W. Zhang, T. Xing, J.-H. Lin, M. Srivastava, R. Gupta, and Z. Zhang, “Accelerating Binarized Convolutional Neural Networks with Software-Programmable FPGAs,” *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays - FPGA '17*, pp. 15–24, 2017. ISBN: 9781450343541.

[51] Y. Li, K. Xu, and H. Yu, “A GPU-Outperforming FPGA Accelerator Architecture for Binary Convolutional Neural Networks,” *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 14, no. 2, pp. 1–16, 2018. arXiv: 1702.06392.

[52] L. Yang, Z. He, and D. Fan, “A Fully Onchip Binarized Convolutional Neural Network FPGA Impelmentation with Accurate Inference,” in *Proceedings of the International Symposium on Low Power Electronics and Design*, pp. 50:1–50:6, 2018.

[53] R. Andri, L. Cavigelli, D. Rossi, and L. Benini, “Yoda NN: An Architecture for Ultralow Power Binary-Weight CNN Acceleration,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 1, pp. 48–60, 2018. arXiv: 1606.05487 ISBN: 0278-0070 VO - PP.

[54] A. A. Bahou, G. Karunaratne, R. Andri, L. Cavigelli, and L. Benini, “XNORBIN: A 95 TOp/s/W Hardware Accelerator for Binary Convolutional Neural Networks,” *CoRR*, vol. abs/1803.0, pp. 1–3, 2018. arXiv: 1803.05849 ISBN: 9781538661024.

[55] F. Conti, P. D. Schiavone, and L. Benini, “XNOR Neural Engine: a Hardware Accelerator IP for 21.6 fJ/op Binary Neural Network Inference,” 2018. arXiv: 1807.03010.

[56] K. Ando, K. Ueyoshi, K. Orimo, H. Yonekawa, S. Sato, H. Nakahara, S. Takamaeda-Yamazaki, M. Ikebe, T. Asai, T. Kuroda, and M. Motomura, “BRein Memory : A Single-Chip Binary / Ternary Reconfigurable in-Memory Deep Neural Network,” *IEEE Journal of Solid-State Circuits*, vol. 53, no. 4, pp. 983–994, 2017.

[57] L. Jiang, M. Kim, W. Wen, and D. Wang, “XNOR-POP: A Processing-in-Memory Architecture for Binary Convolutional Neural Networks in Wide-IO2 DRAMs,” in *2017 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, (Taipei, Taiwan), pp. 1–6, IEEE, 2017. ISBN: 9781509060238.

[58] D. Bankman, L. Yang, B. Moons, M. Verhelst, and B. Murmann, “An Always-On 3.8 μ J/86% CIFAR-10 Mixed-Signal Binary CNN Processor With All Memory on Chip in 28-nm CMOS,” *IEEE Journal of Solid-State Circuits*, vol. 54, pp. 158–172, Jan. 2019.

[59] H. Yang, M. Fritzsche, C. Bartz, and C. Meinel, “BMXNet: An Open-Source Binary Neural Network Implementation Based on MXNet,” in *Proceedings of the 25th ACM international conference on Multimedia*, (Mountain View, California, USA), pp. 1209–1212, Association for Computing Machinery, 2017. arXiv: 1705.09864 ISSN: 16130073.

[60] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, “MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems,” *arXiv preprint arXiv:1512.01274*, 2015. arXiv: 1512.01274 ISBN: 0360-0300.

[61] Y. Hu, J. Zhai, D. Li, Y. Gong, Y. Zhu, W. Liu, L. Su, and J. Jin, “BitFlow: Exploiting Vector Parallelism for Binary Neural Networks on CPU,” in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, (Vancouver, British Columbia, Canada), pp. 244–253, IEEE, 2018. Publisher: IEEE ISBN: 978-1-5386-4368-6.

[62] F. Pedersoli, G. Tzanetakis, and A. Tagliasacchi, “Espresso: Efficient Forward Propagation for BCNNs,” 2017. arXiv: 1705.07175v2.

[63] B. McDanel, S. Teerapittayanon, and H. T. Kung, “Embedded Binarized Neural Networks,” 2017. arXiv: 1709.02260 ISBN: 9780994988614.

[64] D. Wu, J. Li, R. Yin, H. Hsiao, Y. Kim, and J. S. Miguel, “uGEMM : Unary Computing Architecture for GEMM Applications,” *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pp. 377–390, 2020. ISBN: 9781728146614.

[65] S. Mohajer, Z. Wang, and K. Bazargan, “Routing Magic: Performing Computations Using Routing Networks and Voting Logic on Unary Encoded Data,” in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 77–86, 2018.

[66] A. Madhavan, T. Sherwood, and D. Strukov, “Race Logic: A Hardware Acceleration for Dynamic Programming Algorithms,” *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3, pp. 517–528, 2014.

[67] D. Wu and J. S. Miguel, "uSystolic : Byte-Crawling Unary Systolic Array," in *2022 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2022.

[68] B. S. B. Furber, F. Ieee, F. Galluppi, S. Temple, L. A. Plana, and S. M. Ieee, "The SpiNNaker Project," *Proceedings of the IEEE*, vol. 102, no. 5, pp. 652–665, 2014. Publisher: IEEE.

[69] R. Silver, K. Boahen, S. Grillner, N. Kopell, and K. L. Olsen, "Neurotech for Neuroscience: Unifying Concepts, Organizing Principles, and Emerging Tools," *Journal of Neuroscience*, vol. 27, pp. 11807–11819, Oct. 2007.

[70] J. Schemmel, D. Briiderle, A. Griibl, M. Hock, K. Meier, and S. Millner, "A wafer-scale neuromorphic hardware system for large-scale neural modeling," in *Proceedings of 2010 IEEE International Symposium on Circuits and Systems*, (Paris, France), pp. 1947–1950, IEEE, May 2010.

[71] S. W. Moore, P. J. Fox, S. J. Marsh, A. T. Markettos, and A. Mujumdar, "Bluehive - A field-programable custom computing machine for extreme-scale real-time neural network simulation," in *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*, (Toronto, ON), pp. 133–140, IEEE, Apr. 2012.

[72] R. Serrano-Gotarredona, M. Oster, P. Lichtsteiner, A. Linares-Barranco, R. Paz-Vicente, F. Gómez-Rodríguez, L. Camunas-Mesa, R. Berner, M. Rivas-Perez, T. Delbrück, S. C. Liu, R. Douglas, P. Hafliger, G. Jimenez-Moreno, A. Civit Ballcels, T. Serrano-Gotarredona, A. J. Acosta-Jimenez, and B. Linares-Barranco, "CAVIAR: A 45k neuron, 5M synapse, 12G connects/s AER hardware sensory-processing-learning-actuating system for high-speed visual object recognition and tracking," *IEEE Transactions on Neural Networks*, vol. 20, no. 9, pp. 1417–1438, 2009.

[73] M. Bouvier, A. Valentian, T. Mesquida, F. Rummens, M. Reyboz, E. Vianello, and E. Beigne, "Spiking Neural Networks Hardware Implementations and Challenges: A Survey," *ACM Journal on Emerging Technologies in Computing Systems*, vol. 15, pp. 1–35, Apr. 2019.

[74] J. L. Gustafson and I. Yonemoto, "Beating Floating Point at its Own Game: Posit Arithmetic," *Supercomputing frontiers and innovations*, vol. 4, no. 2, pp. 71–86, 2017.

[75] D. Mallasén, R. Murillo, A. A. Del Barrio, G. Botella, L. Piñuel, and M. Prieto, "PERCIVAL: Open-Source Posit RISC-V Core with Quire Capability," *IEEE Transactions on Emerging Topics in Computing*, vol. 10, pp. 1241–1252, July 2022.

[76] M. K. Jaiswal and H. K.-H. So, "PACoGen: A Hardware Posit Arithmetic Core Generator," *IEEE Access*, vol. 7, pp. 74586–74601, 2019.

[77] E. Park, J. Ahn, and S. Yoo, "Weighted-Entropy-based Quantization for Deep Neural Networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 5456–5464, 2017.

[78] T. Kudo, K. Ueyoshi, K. Ando, K. Hirose, R. Uematsu, Y. Oba, M. Ikebe, T. Aasai, M. Motomura, and S. Takamaeda-yamazaki, "Area and Energy Optimization for Bit-Serial Log-Quantized DNN Accelerator with Shared Accumulators," in *2018 IEEE 12th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*, pp. 237–243, 2018.

[79] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "SCNN: An Accelerator for Compressed-sparse Convolutional Neural Networks," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pp. 27–40, 2017. arXiv: 1708.04485 ISSN: 10636897.

[80] B. Hassibi, D. G. Stork, and G. J. Wolff, "Optimal brain surgeon and general network pruning," *IEEE International Conference on Neural Networks - Conference Proceedings*, vol. 1993-Janua, pp. 293–299, 1993. ISBN: 0780309995.

[81] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "EIE: Efficient Inference Engine on Compressed Deep Neural Network," vol. 16, 2016. arXiv: 1602.01528 ISBN: 978-1-4673-8947-1.

[82] S. Han, J. Pool, J. Tran, and W. J. Dally, "Learning both Weights and Connections for Efficient Neural Networks," in *Advances in neural information processing systems*, pp. 1135–1143, 2015. arXiv: 1506.02626 ISSN: 01406736.

[83] S. Han, H. Mao, and W. J. Dally, "Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding," *arXiv preprint arXiv:1510.00149*, pp. 1–14, 2015. arXiv: 1510.00149 ISBN: 0470021438.

[84] R. Sredojevic, S. Cheng, L. Supic, R. Naous, and V. Stojanovic, "Structured Deep Neural Network Pruning via Matrix Pivoting," *arXiv preprint arXiv:1712.01084*, pp. 1–16, 2017. arXiv: 1712.01084.

[85] V. Lebedev and V. Lempitsky, "Fast ConvNets Using Group-wise Brain Damage," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 2554–2564, 2015. arXiv: 1506.02515 ISBN: 978-1-4673-8851-1.

[86] H. Foroosh, M. Tappen, and M. Penksy, "Sparse Convolutional Neural Networks," *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 806–814, 2015. ISBN: 978-1-4673-6964-0.

[87] S. Anwar and W. Sung, "Compact Deep Convolutional Neural Networks With Coarse Pruning," *arXiv preprint arXiv:1610.09639*, 2016. arXiv: 1610.09639.

[88] S. Anwar, K. Hwang, and W. Sung, "Structured Pruning of Deep Convolutional Neural Networks," *ACM Journal on Emerging Technologies in Computing Systems*, vol. 13, no. 3, pp. 1–18, 2017. arXiv: 1512.08571 ISBN: 9781509059904.

[89] H. Wang, Q. Zhang, Y. Wang, and R. Hu, "Structured Deep Neural Network Pruning by Varying Regularization Parameters," *ArXiv preprint: 1804.09461*, vol. 3, 2018. arXiv: 1804.09461.

[90] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen, "Cambricon-X : An Accelerator for Sparse Neural Networks," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–12, 2016.

[91] L. Liang, L. Deng, Y. Zeng, X. Hu, Y. Ji, X. Ma, G. Li, and Y. Xie, "Crossbar-Aware Neural Network Pruning," *IEEE Access*, vol. 6, pp. 58324–58337, 2018. arXiv: 1807.10816.

[92] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "SqueezeNet: AlexNet-level Accuracy with 50x Fewer Parameters and <0.5MB Model Size," *arXiv preprint arXiv:1602.07360*, pp. 1–13, 2016. arXiv: 1602.07360 ISBN: 978-3-319-24552-2.

[93] X. Zhang, X. Zhou, M. Lin, and J. Sun, "ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices," in *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, (Salt Lake City, UT), pp. 6848–6856, IEEE, June 2018.

[94] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications," *arXiv preprint arXiv:1704.04861*, 2017. arXiv: 1704.04861 ISBN: 2004012439.

[95] L. Lai, N. Suda, and V. Chandra, "CMSIS-NN: Efficient Neural Network Kernels for Arm Cortex-M CPUs," *arXiv preprint arXiv:1801.06601*, pp. 1–10, 2018. arXiv: 1801.06601.

[96] A. Kumar, S. Goyal, and M. Varma, "Resource-efficient Machine Learning in 2 KB RAM for the Internet of Things," in *International Conference on Machine Learning*, pp. 1935–1944, 2017.

[97] C. Gupta, A. S. Suggala, A. Goyal, H. V. Simhadri, B. Paranjape, A. Kumar, S. Goyal, R. Udupa, M. Varma, and P. Jain, "ProtoNN: Compressed and Accurate kNN for Resource-scarce Devices," in *International Conference on Machine Learning*, pp. 1331–1340, 2017.

[98] R. David, J. Duke, A. Jain, V. J. Reddi, N. Jeffries, J. Li, N. Kreeger, I. Nappier, M. Natraj, S. Regev, R. Rhodes, T. Wang, and P. Warden, "TensorFlow Lite Micro: Embedded Machine Learning on TinyML Systems," in *Proceedings of Machine Learning and Systems*, pp. 800–811, 2021.

[99] J. Lin, W.-M. Chen, Y. Lin, J. Cohn, C. Gan, and S. Han, "MCUNet: Tiny Deep Learning on IoT Devices," in *Advances in Neural Information Processing Systems*, vol. 33, pp. 11711–11722, 2020.

[100] B. R. Gaines, "Stochastic computing systems," *Advances in Information Systems Science*, vol. 2, pp. 37–172, 1969.

[101] A. Alaghi, *The Logic of Random Pulses: Stochastic Computing.* PhD thesis, University of Michigan, 2015.

[102] A. Alaghi and J. P. Hayes, "Survey of Stochastic Computing," *ACM Transactions on Embedded computing systems (TECS)*, vol. 12, no. 2s, pp. 1–19, 2013.

[103] B. Yuan, Y. Wang, and Z. Wang, "Area-Efficient Scaling-free DFT / FFT Design using Stochastic Computing," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 63, no. 1, pp. 1131–1135, 2016. ISBN: 9781479953417.

[104] R. Hojabr, K. Givaki, S. M. R. Tayaranian, P. Esfahanian, A. Khonsari, D. Rahmati, and M. H. Najafi, "SkippyNN : An Embedded Stochastic-Computing Accelerator for Convolutional Neural Networks," in *2019 56th ACM/IEEE Design Automation Conference (DAC)*, pp. 1–6, 2019.

[105] Z. Li, J. Li, A. Ren, R. Cai, C. Ding, X. Qian, J. Draper, B. Yuan, J. Tang, Q. Qiu, and Others, "HEIF: Highly Efficient Stochastic Computing based Inference Framework for Deep Neural Networks," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 8, pp. 1543–1556, 2018.

[106] A. Ren, J. Li, Z. Li, C. Ding, X. Qian, Q. Qiu, B. Yuan, and Y. Wang, "SC-DCNN: Highly-Scalable Deep Convolutional Neural Network using Stochastic Computing," *ACM SIGPLAN Notices*, vol. 52, no. 4, pp. 405–418, 2017. arXiv: 1611.05939 ISBN: 9781450344654.

[107] H. Sim and J. Lee, "A New Stochastic Computing Multiplier with Application to Deep Convolutional Neural Networks," in *Proceedings of the 54th Annual Design Automation Conference 2017 on - DAC '17*, pp. 1–6, 2017. ISSN: 0738100X.

[108] J. Yu, K. Kim, J. Lee, and K. Choi, "Accurate and Efficient Stochastic Computing Hardware for Convolutional Neural Networks," *2017 IEEE International Conference on Computer Design (ICCD)*, pp. 105–112, 2017. ISBN: 978-1-5386-2254-4.

[109] A. Zhakatayev, S. Lee, H. Sim, and J. Lee, "Sign-Magnitude SC : Getting 10X Accuracy for Free in Stochastic Computing for Deep Neural Networks," in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pp. 1–6, IEEE, 2018.

[110] S. R. Faraji, M. H. Najafi, B. Li, D. J. Lilja, and K. Bazargan, "Energy-Efficient Convolutional Neural Networks with Deterministic Bit-Stream Processing," in *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 1757–1762, EDAA, 2019.

[111] J. A. Dickson, R. D. Mcleod, and H. C. Card, "Stochastic Arithmetic Implementations of Neural Networks with in Situ Learning," in *IEEE International Conference on Neural Networks*, pp. 711–716, 1993.

[112] B. Brown and H. Card, "Stochastic neural computation. I. Computational elements," *IEEE Transactions on Computers*, vol. 50, pp. 891–905, Sept. 2001.

[113] D. Larkin, A. Kinane, V. Muresan, and N. O'Connor, "An Efficient Hardware Architecture for a Neural Network Activation Function Generator," *International Symposium on Neural Networks*, pp. 1319–1327, 2006. ISBN: 3540344829.

[114] T.-H. Chen and J. P. Hayes, "Design of Division Circuits for Stochastic Computing," in *2016 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, (Pittsburgh, PA, USA), pp. 116–121, IEEE, July 2016.

[115] V. T. Lee, A. Alaghi, L. Ceze, and M. Oskin, "Stochastic Synthesis for Stochastic Computing," *arXiv preprint arXiv:1810.04756*, 2018. arXiv: 1810.04756.

[116] Z. Li, A. Ren, J. Li, Q. Qiu, B. Yuan, J. Draper, and Y. Wang, "Structural Design Optimization for Deep Convolutional Neural Networks Using Stochastic Computing," *Proceedings of the 2017 Design, Automation and Test in Europe, DATE 2017*, pp. 250–253, 2017. ISBN: 9783981537093.

[117] V. Govindaraju, C.-H. Ho, T. Nowatzki, J. Chhugani, N. Satish, K. Sankaralingam, and C. Kim, "DySER: Unifying Functionality and Parallelism Specialization for Energy-Efficient Computing," *IEEE Micro*, vol. 32, pp. 38–51, Sept. 2012.

[118] M. Zaheer, G. Guruganesh, A. Dubey, J. Ainslie, C. Alberti, S. Ontanon, P. Pham, A. Ravula, Q. Wang, L. Yang, and A. Ahmed, "Big Bird: Transformers for Longer Sequences," in *Advances in neural information processing systems*, vol. 33, pp. 17283–17297, 2020.

[119] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, and T. Henighan, "Language Models are Few-Shot Learners," in *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[120] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "DianNao: A Small-Footprint High-Throughput Accelerator for Ubiquitous Machine-Learning," *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*, pp. 269–284, 2014. ISBN: 9781450323055.

[121] J. Lee, C. Kim, S. Kang, D. Shin, S. Kim, and H.-j. Yoo, "UNPU: A 50.6TOPS/W Unified Deep Neural Network Accelerator with 1b-to-16b Fully-Variable Weight Bit-Precision," in *2018 IEEE International Solid - State Circuits Conference - (ISSCC)*, pp. 218–220, IEEE, 2018.

[122] S. Sharify, A. D. Lascorz, M. Mahmoud, M. Nikolic, K. Siu, D. M. Stuart, Z. Poulos, and A. Moshovos, "Laconic deep learning inference acceleration," *Proceedings - International Symposium on Computer Architecture*, pp. 304–317, 2019. ISBN: 9781450366694.

[123] J. Albericio, P. Judd, A. Delmas, S. Sharify, and A. Moshovos, "Bit-Pragmatic Deep Neural Network Computing," *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 382–394, 2017. ISBN: 9781450349529.

[124] H. Sharma, J. Park, and B. Chau, "Bit Fusion : Bit-Level Dynamically Composable Architecture for Accelerating Deep Neural Networks," *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pp. 764–775, 2018. Publisher: IEEE.

[125] A. Aimar, H. Mostafa, E. Calabrese, A. Rios-navarro, R. Tapiador-morales, I.-a. Lungu, M. B. Milde, F. Corradi, A. Linares-barranco, S.-c. Liu, and T. Delbruck, "NullHop : A Flexible Convolutional Neural Network Accelerator Based on Sparse Representations of Feature Maps," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 30, no. 3, pp. 644–656, 2019. Publisher: IEEE.

[126] C. Deng, S. Yang, S. Liao, and B. Qian, XuehaiYuan, "GoSPA : An Energy-efficient High-performance Globally Optimized SParse Convolutional Neural Network Accelerator," in *International Symposium on Computer Architecture (ISCA)*, pp. 1110–1123, 2021.

[127] A. Gondimalla, N. Chesnut, M. Thottethodi, and T. Vijaykumar, "SparTen: A Sparse Tensor Accelerator for Convolutional NeuralNetworks," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 151–165, 2019. ISSN: 18697720.

[128] N. Srivastava, H. Jin, S. Smith, H. Rong, D. Albonesi, and Z. Zhang, "Tensaurus : A Versatile Accelerator for Mixed Sparse-Dense Tensor Computations," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 689–702, 2020.

[129] H. Kwon, P. Chatarasi, M. Pellauer, A. Parashar, V. Sarkar, and T. Krishna, "Understanding reuse, performance, and hardware cost of DNN dataflows: A data-centric approach using MAESTRO," in *Proceedings of the Annual International Symposium on Microarchitecture, MICRO*, pp. 754–768, 2019. arXiv: 1805.02566v6 ISSN: 10724451.

[130] H. Kwon, A. Samajdar, and T. Krishna, "MAERI: Enabling Flexible Dataflow Mapping over DNN Accelerators via Reconfigurable Interconnects," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 461–475, 2018. ISSN: 15232867.

[131] S. Venkataramani, P. Dubey, A. Raghunathan, A. Ranjan, S. Banerjee, D. Das, S. Avancha, A. Jagannathan, A. Durg, D. Nagaraj, and B. Kaul, "ScaleDeep: A Scalable Compute Architecture for Learning and Evaluating Deep Networks," in *Proceedings of the 44th Annual International Symposium on Computer Architecture - ISCA '17*, pp. 13–26, 2017. ISSN: 10636897.

[132] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar, "ISAAC: A Convolutional Neural Network Accelerator with In-Situ Analog Arithmetic in Crossbars," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 14–26, 2016.

[133] S. Li, D. Niu, K. T. Malladi, H. Zheng, B. Brennan, Y. Xie, and H. Zheng, "DRISA : A DRAM-based Reconfigurable In-Situ Accelerator," in *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, vol. 14, pp. 288–301, 2017. ISSN: 10724451.

[134] S. Dutta, S. A. Siddiqui, B. Felix, L. Liu, C. A. Ross, and M. A. Baldo, "A Logic-in-Memory Design with 3-Terminal Magnetic Tunnel Junction Function Evaluators for Convolutional Neural Networks," pp. 83–88, 2017. ISBN: 9781509060375.

[135] S. Gupta, M. Imani, J. Sim, A. Huang, F. Wu, M. H. Najafi, T. Rosing, S. Diego, and L. Jolla, "SCRIMP : A General Stochastic Computing Architecture using ReRAM in-Memory Processing," in *2020 Design, Automation \& Test in Europe Conference \& Exhibition (DATE)*, pp. 1598–1601, 2020.

[136] H. Jia, M. Ozatay, Y. Tang, H. Valavi, R. Pathak, J. Lee, and N. Verma, "A Programmable Neural-Network Inference Accelerator Based on Scalable In-Memory Computing," in *2021 IEEE International Solid-State Circuits Conference-(ISSCC)*, pp. 236–237, 2021.

[137] S. Li, C. Xu, Q. Zou, J. Zhao, Y. Lu, and Y. Xie, "Pinatubo: A Processing-in-Memory Architecture for Bulk Bitwise Operations in Emerging Non-volatile Memories," *Proceedings of the 53rd Annual Design Automation Conference on - DAC '16*, pp. 1–6, 2016. ISBN: 9781450342360.

[138] X. Ma, L. Chang, S. Li, L. Deng, Y. Ding, and Y. Xie, “In-Memory Multiplication Engine with SOT-MRAM Based Stochastic Somputing,” *arXiv preprint arXiv:1809.08358*, 2018. arXiv: 1809.08358v1.

[139] STMicroelectronics, “STM32 Nucleo Boards.”

[140] “Raspberry Pi.”

[141] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio, “Binarynet: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1,” 2016. arXiv: 1602.02830 ISBN: 9781510829008.

[142] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, “Automatic differentiation in PyTorch,”

[143] S. Ioffe and C. Szegedy, “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift,” in *International Conference on Machine Learning*, pp. 448–456, Mar. 2015.

[144] A. Buluc, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson, “Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks,” in *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, (Calgary AB Canada), pp. 233–244, ACM, Aug. 2009.

[145] P. Warden, “Speech Commands: A Dataset for Limited-Vocabulary Speech Recognition,” Apr. 2018. Number: arXiv:1804.03209 arXiv:1804.03209 [cs].

[146] R. C. Prim, “Shortest Connection Networks And Some Generalizations,” *Bell System Technical Journal*, vol. 36, pp. 1389–1401, Nov. 1957.

[147] K. Chellapilla, S. Puri, and P. Simard, “High Performance Convolutional Neural Networks for Document Processing,” in *Tenth International Workshop on Frontiers in Handwriting Recognition*, 2006.

[148] ARM, “NEON.”

[149] STMicroelectronics, “STM32 Nucleo-144 board.”

[150] A. Limited, “ARMv7-M Architecture Reference Manual.”

[151] A. Limited, “ARMv6-M Architecture Reference Manual.”

[152] Y. LeCun, L. Bottou, and Y. Bengio, “Gradient-Based Learning Applied to Document Recognition,” *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[153] D. P. Kingma and J. Ba, “Adam: A Method for Stochastic Optimization,” Jan. 2017. Number: arXiv:1412.6980 arXiv:1412.6980 [cs].

[154] A. Limited, "ARM Compute Library."

[155] W. Mula, N. Kurz, and D. Lemire, "Faster Population Counts Using AVX2 Instructions," *Computer Journal*, vol. 61, no. 1, pp. 111–120, 2018. arXiv: 1611.07612 ISBN: 5555555555.

[156] F. Li, B. Zhang, and B. Liu, "Ternary Weight Networks," *arXiv preprint arXiv:1605.04711*, no. Nips, 2016. arXiv: 1605.04711.

[157] H. Alemdar, V. Leroy, A. Prost-Boucle, and F. Petrot, "Ternary Neural Networks for Resource-Efficient AI Applications," *Proceedings of the International Joint Conference on Neural Networks*, pp. 2547–2554, 2017. arXiv: 1609.00222 ISBN: 9781509061815.

[158] N. Mellempudi, A. Kundu, D. Mudigere, D. Das, B. Kaul, and P. Dubey, "Ternary Neural Networks with Fine-Grained Quantization," *arXiv preprint arXiv:1705.01462*, 2017. arXiv: 1705.01462.

[159] A. Kundu, K. Banerjee, N. Mellempudi, D. Mudigere, D. Das, B. Kaul, and P. Dubey, "Ternary Residual Networks," *arXiv preprint arXiv:1707.04679*, pp. 1–16, 2017. arXiv: 1707.04679.

[160] H. Yonekawa, S. Sato, and H. Nakahara, "A Ternary Weight Binary Input Convolutional Neural Network: Realization on the Embedded Processor," *2018 IEEE 48th International Symposium on Multiple-Valued Logic (ISMVL)*, pp. 174–179, 2018. Publisher: IEEE ISBN: 978-1-5386-4464-5.

[161] Z. He, B. Gong, and D. Fan, "Optimize Deep Convolutional Neural Network with Ternarized Weights and High Accuracy," in *2019 IEEE Winter Conference on Applications of Computer Vision (WACV)*, pp. 913–921, 2019. arXiv: 1807.07948v1.

[162] J. H. Lin, T. Xing, R. Zhao, Z. Zhang, M. Srivastava, Z. Tu, and R. K. Gupta, "Binarized Convolutional Neural Networks with Separable Filters for Efficient Hardware Acceleration," *IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, vol. 2017-July, no. 1, pp. 344–352, 2017. arXiv: 1707.04693 ISBN: 9781538607336.

[163] M. P. Heinrich, M. Blendowski, and O. Oktay, "TernaryNet: faster deep model inference without GPUs for medical 3D segmentation using sparse and binary convolutions," *International Journal of Computer Assisted Radiology and Surgery*, pp. 1–10, 2018. arXiv: 1801.09449.

[164] L. Deng, P. Jiao, J. Pei, Z. Wu, and G. Li, "GXNOR-Net: Training deep neural networks with ternary weights and activations without full-precision memory under a unified discretization framework," *Neural Networks*, vol. 100, pp. 49–58, 2018. arXiv: 1705.09283.

[165] C. Li, H. Farkhoor, R. Liu, and J. Yosinski, "Measuring the intrinsic dimension of objective landscapes," *arXiv*, 2018. arXiv: 1804.08838.

[166] S. Li, A. O. Glova, X. Hu, P. Gu, D. Niu, K. T. Malladi, H. Zheng, B. Brennan, and Y. Xie, "SCOPE: A stochastic computing engine for DRAM-based in-situ accelerator," *Proceedings of the Annual International Symposium on Microarchitecture, MICRO*, vol. 2018-Octob, pp. 696–709, 2018. ISBN: 9781538662403.

[167] M. H. Ionica and D. Gregg, "The Movidius Myriad Architecture's Potential for Scientific Computing," *IEEE Micro*, vol. 35, no. 1, pp. 6–14, 2015.

[168] D. D. Lin, S. S. Talathi, T. G. Com, V. S. Annapureddy, and S. G. Com, "Fixed Point Quantization of Deep Convolutional Networks," in *International Conference on Machine Learning*, pp. 2849–2858, 2016. arXiv: 1511.06393v3.

[169] S. Anwar, K. Hwang, and W. Sung, "Fixed Point Optimization of Deep Convolutional Neural Networks for Object Recognition," in *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 1131–1135, IEEE, 2015.

[170] V. T. Lee, A. Alaghi, R. Pamula, V. S. Sathe, L. Ceze, and M. Oskin, "Architecture Considerations for Stochastic Computing Accelerators," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2277–2289, 2018. Publisher: IEEE.

[171] H. Sim, D. Nguyen, J. Lee, and K. Choi, "Scalable Stochastic-Computing Accelerator for Convolutional Neural Networks," in *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 696–701, IEEE, 2017.

[172] V. T. Lee, A. Alaghi, J. P. Hayes, V. Sathe, and L. Ceze, "Energy-Efficient Hybrid Stochastic-Binary Neural Networks for Near-Sensor Computing," in *Design, Automation \& Test in Europe Conference \& Exhibition (DATE), 2017*, pp. 13–18, 2017.

[173] Z. Li, A. Ren, J. Li, Q. Qiu, B. Yuan, J. Draper, and Y. Wang, "Structural Design Optimization for Deep Convolutional Neural Networks using Stochastic Computing," *Proceedings of the 2017 Design, Automation & Test in Europe Conference & Exhibition*, no. c, pp. 250–253, 2017. ISBN: 9783981537086.

[174] A. Krizhevsky, "Learning Multiple Layers of Features from Tiny Images,"

[175] H. Ichihara, S. Ishii, D. Sunamori, T. Iwagaki, and T. Inoue, "Compact and Accurate Stochastic Circuits with Shared Random Number Sources," in *2014 IEEE 32nd International Conference on Computer Design (ICCD)*, pp. 361–366, IEEE, 2014.

[176] J. T. Springenberg, A. Dosovitskiy, T. Brox, and M. Riedmiller, "Striving for Simplicity: The All Convolutional Net," in *ICLR (workshop track)*, pp. 1–14, 2015. arXiv: 1412.6806v3.

[177] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "CACTI 6.0 : A Tool to Model Large Caches," *HP laboratories*, vol. 27, no. HPL-2009-85, p. 28, 2009.

[178] M. Gao and M. Horowitz, "TETRIS: Scalable and Efficient Neural Network Acceleration with 3D Memory," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 751–764, 2017.

[179] K. Guo, L. Sui, J. Qiu, J. Yu, J. Wang, S. Yao, S. Han, Y. Wang, and H. Yang, "Angel-Eye : A Complete Design Flow for Mapping CNN Onto Embedded FPGA," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 1, pp. 35–47, 2017. Publisher: IEEE.

[180] K. Guo, L. Sui, J. Qiu, S. Yao, S. Han, Y. Wang, and H. Yang, "From Model to FPGA : Software-Hardware Co-Design for Efficient Neural Network Acceleration," in *2016 IEEE Hot Chips 28 Symposium (HCS)*, pp. 1–27, IEEE, 2016.

[181] H. Wang, X. Zhang, D. Kong, and G. Lu, "Convolutional Neural Network Accelerator on FPGA," in *2019 IEEE International Conference on Integrated Circuits, Technologies and Applications (ICTA)*, pp. 61–62, 2019.

[182] X. Hu, Y. Zeng, Z. Li, X. I. N. Zheng, S. Cai, and X. Xiong, "A Resources-Efficient Configurable Accelerator for Deep Convolutional Neural Networks," *IEEE Access*, vol. 7, pp. 72113–72124, 2019. Publisher: IEEE.

[183] A. Agrawal, S. K. Lee, J. Silberman, M. Ziegler, M. Kang, S. Venkataramani, N. Cao, B. Fleischer, M. Guillorn, M. Cohen, S. Mueller, J. Oh, M. Lutz, J. Jung, S. Koswatta, C. Zhou, V. Zalani, J. Bonanno, R. Casatuta, C.-Y. Chen, J. Choi, H. Haynie, A. Herbert, R. Jain, M. Kar, K.-H. Kim, Y. Li, Z. Ren, S. Rider, M. Schaal, K. Schelm, M. Scheuermann, X. Sun, H. Tran, N. Wang, W. Wang, X. Zhang, V. Shah, C. Brian, V. Srinivasan, P.-F. Lu, S. Shukla, L. Chang, and K. Gopalakrishnan, "A 7nm 4-Core AI Chip with 25.6TFLOPS Hybrid FP8 Training, 102.4TOPS INT4 Inference and Workload-Aware Throttling," in *2021 IEEE International Solid-State Circuits Conference-(ISSCC)*, pp. 144–145, 2021.

[184] J. Park, J. Lee, and D. Jeon, "7.6 A 65nm 236.5nJ/Classification Neuromorphic Processor with 7.5% Energy Overhead On-Chip Learning Using Direct Spike-Only Feedback," in *2019 IEEE International Solid- State Circuits Conference - (ISSCC)*, (San Francisco, CA, USA), pp. 140–142, IEEE, Feb. 2019.

[185] C. Torres-Huitzil and B. Girau, "Fault and Error Tolerance in Neural Networks: A Review," *IEEE Access*, vol. 5, pp. 17322–17341, 2017.

[186] Y. Liu, S. Liu, Y. Wang, F. Lombardi, and J. Han, "A Stochastic Computational Multi-Layer Perceptron with Backward Propagation," *IEEE Transactions on Computers*, vol. 67, pp. 1273–1286, Sept. 2018.

[187] M. H. Najafi, D. Jenson, D. J. Lilja, and M. D. Riedel, "Performing Stochastic Computation Deterministically," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 12, pp. 2925–2938, 2019. Publisher: IEEE.

[188] B. Li, M. H. Najafi, B. Yuan, and D. J. Lilja, "Quantized neural networks with new stochastic multipliers," in *2018 19th International Symposium on Quality Electronic Design (ISQED)*, (Santa Clara, CA), pp. 376–382, IEEE, Mar. 2018.

[189] A. Ren, Z. Li, Y. Wang, Q. Qiu, and B. Yuan, "Designing reconfigurable large-scale deep learning systems using stochastic computing," in *2016 IEEE International Conference on Rebooting Computing (ICRC)*, (San Diego, CA, USA), pp. 1–7, IEEE, Oct. 2016.

[190] N. Nedjah and L. de Macedo Mourelle, "Stochastic reconfigurable hardware for neural networks," in *Euromicro Symposium on Digital System Design, 2003. Proceedings.*, (Belek-Antalya, Turkey), pp. 438–442, IEEE, 2003.

[191] V. Canals, A. Morro, A. Oliver, M. L. Alomar, and J. L. Rosselló, "A New Stochastic Computing Methodology for Efficient Neural Network Implementation," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 27, no. 3, pp. 551–564, 2015. Publisher: IEEE.

[192] S. Toral, J. Quero, and L. Franquelo, "Stochastic pulse coded arithmetic," in *2000 IEEE International Symposium on Circuits and Systems. Emerging Technologies for the 21st Century. Proceedings (IEEE Cat No.00CH36353)*, vol. 1, (Geneva, Switzerland), pp. 599–602, Presses Polytech. Univ. Romandes, 2000.

[193] A. Ardakani, C. Condo, and W. J. Gross, "Sparsely-Connected Neural Networks: Towards Efficient VLSI Implementation of Deep Neural Networks," *arXiv preprint arXiv:1611.01427*, 2016. arXiv: 1611.01427.

[194] C. Lammie and M. R. Azghadi, "Stochastic Computing for Low-Power and High-Speed Deep Learning on FPGA," in *2019 IEEE International Symposium on Circuits and Systems (ISCAS)*, (Sapporo, Japan), pp. 1–5, IEEE, May 2019.

[195] Y. Wang, J. Lin, and Z. Wang, "FPAP : A Folded Architecture for Efficient Computing of Convolutional Neural Networks," in *2018 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pp. 503–508, IEEE, 2018.

[196] A. Delm, S. Sharify, P. Judd, and A. Moshovos, "Tartan : Accelerating Fully-Connected and Convolutional Layers in Deep Learning Networks by Exploiting Numerical Precision Variability," *arXiv preprint arXiv:1707.09068*, 2017. arXiv: 1707.09068v1.

[197] C. Eckert, X. Wang, J. Wang, A. Subramaniyan, R. Iyer, D. Sylvester, D. Blaauw, and R. Das, "Neural Cache : Bit-Serial In-Cache Acceleration of Deep Neural Networks," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pp. 383–396, 2018.

[198] S. Sharify, A. D. Lascorz, K. Siu, P. Judd, and A. Moshovos, "Loom: Exploiting Weight and Activation Precisions to Accelerate Convolutional Neural Networks," in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pp. 1–6, 2018.

[199] B. Moons and M. Verhelst, "A 0.3–2.6 TOPS/W Precision-Scalable Processor for Real-Time Large-Scale ConvNets," in *2016 IEEE Symposium on VLSI Circuits (VLSI-Circuits)*, pp. 1–2, IEEE, 2016.

[200] B. Moons, R. Uytterhoeven, W. Dehaene, and M. Verhelst, "14.5 Envision: A 0.26-to-10TOPS/W subword-parallel dynamic-voltage-accuracy-frequency-scalable Convolutional Neural Network processor in 28nm FDSOI," in *2017 IEEE International Solid-State Circuits Conference (ISSCC)*, pp. 246–247, IEEE, 2017.

[201] M. H. Najafi, D. J. Lilja, M. D. Riedel, and K. Bazargan, "Polysynchronous Clocking : Exploiting the Skew Tolerance of Stochastic Circuits," *IEEE Transactions on Computers*, vol. 66, no. 10, pp. 1734–1746, 2017.

[202] Z. Yawen, Z. Xinyue, S. Jiahao, W. Yuan, H. Ru, and W. Runsheng, "Parallel Convolutional Neural Network (CNN) Accelerators Based on Stochastic Computing," in *2019 IEEE International Workshop on Signal Processing Systems (SiPS)*, pp. 19–24, 2019.

[203] B. Li, M. H. Najafi, and D. J. Lilja, "Low-Cost Stochastic Hybrid Multiplier for Quantized Neural Networks," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 15, no. 2, 2019.

[204] Z. Li, J. Li, A. Ren, C. Ding, J. Draper, Q. Qiu, B. Yuan, and Y. Wang, "Towards Budget-Driven Hardware Optimization for Deep Convolutional Neural Networks using Stochastic Computing," in *2018 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pp. 28–33, 2018.

[205] X. Ma, Y. Zhang, G. Yuan, A. Ren, Z. Li, J. Han, J. Hu, and Y. Wang, "An Area and Energy Efficient Design of Domain-Wall Memory-Based Deep Convolutional Neural Networks Using Stochastic Computing," in *2018 19th International Symposium on Quality Electronic Design (ISQED)*, pp. 314–321, IEEE, 2018.

[206] A. Mondal and A. Srivastava, "Data Driven Optimizations for MTJ based Stochastic Computing," *arXiv preprint arXiv:1804.03228*.

[207] M. W. Daniels, A. Madhavan, P. Talatchian, A. Mizrahi, and M. D. Stiles, “Energy-Efficient Stochastic Computing with Superparamagnetic Tunnel Junctions,” *Physical Review Applied*, vol. 13, no. 3, p. 034016, 2020. Publisher: American Physical Society.

[208] S. Wang, S. Pal, T. Li, A. Pan, C. Grezes, P. Khalili-Amiri, K. L. Wang, and P. Gupta, “Hybrid VC-MTJ/CMOS Non-volatile Stochastic Logic for Efficient Computing,” in *Design, Automation \& Test in Europe Conference \& Exhibition (DATE), 2017*, vol. 10, pp. 1438–1443, 2017.

[209] F. Neugebauer, I. Polian, and J. P. Hayes, “S-box-Based Random Number Generation for Stochastic Computing,” *Microprocessors and Microsystems*, vol. 61, no. May, pp. 316–326, 2018. Publisher: Elsevier.

[210] S. Liu and J. Han, “Energy Efficient Stochastic Computing with Sobol Sequences,” in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 650–653, EDAA, 2017.

[211] K. Kim, J. Lee, and K. Choi, “Approximate De-randomizer for Stochastic Circuits,” in *2015 International SoC Design Conference (ISOCC)*, pp. 123–124, 2015.

[212] K. Kim, J. Lee, and K. Choi, “Approximate de-randomizer for stochastic circuits,” *ISOCC 2015 - International SoC Design Conference: SoC for Internet of Everything (IoE)*, pp. 123–124, 2016. Publisher: IEEE ISBN: 9781467393089.

[213] M. O’Connor, N. Chatterjee, D. Lee, J. Wilson, A. Agrawal, S. W. Keckler, and W. J. Dally, “Fine-grained DRAM: energy-efficient DRAM for extreme bandwidth systems,” in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, (Cambridge Massachusetts), pp. 41–54, ACM, Oct. 2017.

[214] A. Stillmaker and B. Baas, “Scaling equations for the accurate prediction of CMOS device performance from 180 nm to 7 nm,” *Integration*, vol. 58, no. January, pp. 74–81, 2017. Publisher: Elsevier B.V.

[215] G. Zhong, A. Dubey, T. Cheng, and T. Mitra, “Synergy: A HW/SW framework for high throughput CNNs on embedded heterogeneous SoC,” *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 18, no. 2, pp. 1–23, 2019.

[216] P. Mattson, C. Cheng, C. Coleman, G. Diamos, P. Micikevicius, D. Patterson, H. Tang, G. Y. Wei, P. Bailis, V. Bittorf, D. Brooks, D. Chen, D. Dutta, U. Gupta, K. Hazelwood, A. Hock, X. Huang, B. Jia, D. Kang, D. Kanter, N. Kumar, J. Liao, G. Ma, D. Narayanan, T. Oguntebi, G. Pekhimenko, L. Pentecost, V. J. Reddi, T. Robie, T. S. John, C. J. Wu, L. Xu, C. Young, and M. Zaharia, “The A100 Datacenter GPU and Ampere Architecture,” in *2021 IEEE International Solid- State Circuits Conference (ISSCC)*, pp. 48–50, 2021. arXiv: 1910.01500 ISSN: 23318422.

[217] Z. Gong, H. Ji, C. W. Fletcher, C. J. Hughes, S. Baghsorkhi, and J. Torrellas, "Save: Sparsity-aware vector engine for accelerating DNN training and inference on CPUs," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 796–810, 2020. ISSN: 10724451.

[218] M. Mahmoud, I. Edo, A. H. Zadeh, O. Mohamed Awad, G. Pekhimenko, J. Albericio, and A. Moshovos, "Tensordash: Exploiting sparsity to accelerate deep neural network training," in *Proceedings of the Annual International Symposium on Microarchitecture, MICRO*, pp. 781–795, 2020. arXiv: 2009.00748v1 ISSN: 10724451.

[219] A. D. Lascorz, P. Judd, D. M. Stuart, Z. Poulos, M. Mahmoud, S. Sharify, M. Nikolic, K. Siu, and A. Moshovos, "Bit-Tactical: A Software/Hardware Approach to Exploiting Value and Bit Sparsity in Neural Networks," *International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS*, pp. 749–763, 2019. ISBN: 9781450362405.

[220] K. Hegde, H. Asghari-Moghaddam, M. Pellauer, N. Crago, A. Jaleel, E. Solomonik, J. Emer, and C. W. Fletcher, "ExTensor: An accelerator for sparse tensor algebra," in *Proceedings of the Annual International Symposium on Microarchitecture, MICRO*, pp. 319–333, 2019. ISSN: 10724451.

[221] S. Pal, J. Beaumont, D. H. Park, A. Amarnath, S. Feng, C. Chakrabarti, H. S. Kim, D. Blaauw, T. Mudge, and R. Dreslinski, "OuterSPACE: An Outer Product Based Sparse Matrix Multiplication Accelerator," *Proceedings - International Symposium on High-Performance Computer Architecture*, vol. 2018-Febru, pp. 724–736, 2018. Publisher: IEEE ISBN: 9781538636596.

[222] N. Srivastava, H. Jin, J. Liu, D. Albonesi, and Z. Zhang, "Matraptor: A sparse-sparse matrix multiplication accelerator based on row-wise product," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 766–780, 2020. ISSN: 10724451.

[223] V. Sehwag, N. Prasad, and I. Chakrabarti, "A Parallel Stochastic Number Generator with Bit Permutation Networks," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 65, no. 2, pp. 231–235, 2018. Publisher: IEEE ISBN: 0010101111000.

[224] V. K. Chippa, S. Venkataramani, K. Roy, and A. Raghunathan, "StoRM: A Stochastic Recognition and Mining processor," in *Proceedings of the 2014 International Symposium on Low Power Electronics and Design*, pp. 39–44, 2014. ISSN: 15334678.

[225] E. G. Cota, P. Mantovani, G. Di Guglielmo, and L. P. Carloni, "An analysis of accelerator coupling in heterogeneous architectures," in *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1–6, 2015. ISSN: 0738100X.

[226] Y. Jia, *Learning Semantic Image Representations at a Large Scale*. University of California, Berkeley, 2014. Publication Title: Ph.D. dissertation, EECS Department, University of California, Berkeley.

[227] A. Vasudevan, A. Anderson, and D. Gregg, "Parallel Multi Channel convolution using General Matrix Multiplication," *Proceedings of the International Conference on Application-Specific Systems, Architectures and Processors*, pp. 19–24, 2017. arXiv: 1704.04428 ISBN: 9781509048250.

[228] "DNNSim."

[229] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: Ineffectual-Neuron-Free Deep Neural Network Computing," *Proceedings of the 43rd International Symposium on Computer Architecture*, pp. 1–13, 2016. ISBN: 9781467389471.

[230] X. Zhou, Z. Du, Q. Guo, S. Liu, C. Liu, C. Wang, X. Zhou, L. Li, T. Chen, and Y. Chen, "Cambricon-S: Addressing irregularity in sparse neural networks through a cooperative software/hardware approach," in *Proceedings of the Annual International Symposium on Microarchitecture, MICRO*, pp. 15–28, IEEE, 2018. ISSN: 10724451.

[231] G. Gallego, T. Delbrück, G. Orchard, C. Bartolozzi, B. Taba, A. Censi, S. Leutenegger, A. Davison, J. Conradt, K. Daniilidis, and D. Scaramuzza, "Event-based vision: A survey," *arXiv preprint arXiv:1904.08405*, 2019. arXiv: 1904.08405.

[232] H. Chen, Q. Wu, Y. Liang, X. Gao, and H. Wang, "Asynchronous Tracking-by-Detection on Adaptive Time Surfaces for Event-based Object Tracking," in *Proceedings of the 27th ACM International Conference on Multimedia*, pp. 473–481, 2019.

[233] D. Gehrig, H. Rebecq, G. Gallego, and D. Scaramuzza, "EKLT: Asynchronous Photometric Feature Tracking Using Events and Frames," *International Journal of Computer Vision*, vol. 128, no. 3, pp. 601–618, 2020. Publisher: Springer US.

[234] R. Jiang, X. Mou, S. Shi, Y. Zhou, Q. Wang, M. Dong, and S. Chen, "Object Tracking on Event Cameras with Offline-Online Learning," *CAAI Transactions on Intelligence Technology*, pp. 1–7, 2020.

[235] A. Linares-Barranco, F. Gomez-Rodriguez, V. Villanueva, L. Longinotti, and T. Delbruck, "A USB3.0 FPGA event-based filtering and tracking framework for dynamic vision sensors," *Proceedings - IEEE International Symposium on Circuits and Systems*, vol. 2015-July, pp. 2417–2420, 2015. ISBN: 9781479983919.

[236] C. Scheerlinck, N. Barnes, and R. Mahony, "Asynchronous Spatial Image Convolutions for Event Cameras," *IEEE Robotics and Automation Letters*, vol. 4, no. 2, pp. 816–822, 2019.

[237] H. Rebecq, T. Horstschaefer, G. Gallego, and D. Scaramuzza, "EVO: A Geometric Approach to Event-Based 6-DOF Parallel Tracking and Mapping in Real Time," *IEEE Robotics and Automation Letters*, vol. 2, no. 2, pp. 593–600, 2017. Publisher: IEEE.

[238] C. Scheerlinck, N. Barnes, and R. Mahony, "Continuous-Time Intensity Estimation Using Event Cameras," in *Computer Vision – ACCV 2018* (C. Jawahar, H. Li, G. Mori, and K. Schindler, eds.), vol. 11365, pp. 308–324, Cham: Springer International Publishing, 2019.

[239] H. Patel, C. Iaboni, D. Lobo, J.-w. Choi, and P. Abichandani, "Event Camera Based Real-Time Detection and Tracking of Indoor Ground Robots," *arXiv preprint arXiv:2102.11916*, pp. 1–12, 2021. arXiv: 2102.11916.

[240] P. Bardow, A. J. Davison, and S. Leutenegger, "Simultaneous Optical Flow and Intensity Estimation from an Event Camera," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, (Las Vegas, NV, USA), pp. 884–892, IEEE, June 2016.

[241] C. Brandli, L. Muller, and T. Delbruck, "Real-time, high-speed video decompression using a frame- and event-based DAVIS sensor," in *2014 IEEE International Symposium on Circuits and Systems (ISCAS)*, (Melbourne VIC, Australia), pp. 686–689, IEEE, June 2014.

[242] C. Reinbacher, G. Munda, and T. Pock, "Real-time panoramic tracking for event cameras," in *2017 IEEE International Conference on Computational Photography (ICCP)*, (Stanford, CA, USA), pp. 1–9, IEEE, May 2017.

[243] E. Mueggler, G. Gallego, H. Rebecq, and D. Scaramuzza, "Continuous-Time Visual-Inertial Odometry for Event Cameras," *IEEE Transactions on Robotics*, vol. 34, pp. 1425–1440, Dec. 2018.

[244] A. Linares-Barranco, A. Rios-Navarro, S. Canas-Moreno, E. Piñero-Fuentes, R. Tapiador-Morales, and T. Delbruck, "Dynamic Vision Sensor integration on FPGA-based CNN accelerators for high-speed visual classification," in *International Conference on Neuromorphic Systems*, Association for Computing Machinery, July 2021. arXiv: 1905.07419.

[245] J. Yang, T. Li, W. Romaszkan, P. Gupta, and S. Pamarti, "A 65nm 8-bit All-Digital Stochastic-Compute-In-Memory Deep Learning Processor," in *2022 IEEE Asian Solid-State Circuits Conference (A-SSCC)*, (Taipei, Taiwan), pp. 10–11, IEEE, Nov. 2022.

[246] H. Jia, M. Ozatay, Y. Tang, H. Valavi, R. Pathak, J. Lee, and N. Verma, "A Programmable Neural-Network Inference Accelerator Based on Scalable In-Memory Computing," in *2021 IEEE International Solid- State Circuits Conference (ISSCC)*, (San Francisco, CA, USA), pp. 236–238, IEEE, Feb. 2021.

[247] C. Posch, T. Serrano-Gotarredona, B. Linares-Barranco, and T. Delbruck, "Retinomorphic Event-Based Vision Sensors: Bioinspired Cameras With Spiking Output," *Proceedings of the IEEE*, vol. 102, pp. 1470–1484, Oct. 2014. Conference Name: Proceedings of the IEEE.

[248] T. Delbruck and M. Lang, "Robotic Goalie with 3ms Reaction Time at 4% CPU Load Using Event-Based Dynamic Vision Sensor," *Frontiers in neuroscience*, vol. 7, p. 223, Nov. 2013.

[249] A. Glover and C. Bartolozzi, "Event-driven ball detection and gaze fixation in clutter," in *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 2203–2208, Oct. 2016. ISSN: 2153-0866.

[250] M. Litzenberger, B. Kohn, A. Belbachir, N. Donath, G. Gritsch, H. Garn, C. Posch, and S. Schraml, "Estimation of Vehicle Speed Based on Asynchronous Data from a Silicon Retina Optical Sensor," *2006 IEEE Intelligent Transportation Systems Conference*, pp. 653–658, 2006. Conference Name: 2006 IEEE Intelligent Transportation Systems Conference ISBN: 9781424400935 Place: Toronto, ON, Canada Publisher: IEEE.

[251] H. Kim, S. Leutenegger, and A. J. Davison, "Real-Time 3D Reconstruction and 6-DoF Tracking with an Event Camera," in *Computer Vision–ECCV 2016: 14th European Conference, Amsterdam, The Netherlands, October 11-14, 2016, Proceedings, Part VI 14* (B. Leibe, J. Matas, N. Sebe, and M. Welling, eds.), vol. 9910, (Cham), pp. 349–364, Springer International Publishing, 2016. Book Title: Computer Vision – ECCV 2016 Series Title: Lecture Notes in Computer Science.

[252] A. Rosinol, H. Rebecq, T. Horstschaefer, and D. Scaramuzza, "Ultimate SLAM? Combining Events, Images, and IMU for Robust Visual SLAM in HDR and High Speed Scenarios," *IEEE Robotics and Automation Letters*, vol. PP, pp. 1–1, Jan. 2018.

[253] H. Rebecq, T. Horstschaefer, G. Gallego, and D. Scaramuzza, "EVO: A Geometric Approach to Event-Based 6-DOF Parallel Tracking and Mapping in Real Time," *IEEE Robotics and Automation Letters*, vol. 2, pp. 593–600, Apr. 2017.

[254] C. Brändli, R. Berner, M. Yang, S.-C. Liu, and T. Delbruck, "A 240 × 180 130 dB 3 µs Latency Global Shutter Spatiotemporal Vision Sensor," *Solid-State Circuits, IEEE Journal of*, vol. 49, pp. 2333–2341, Oct. 2014.

[255] T. Delbruck and P. Lichtsteiner, "Fast sensory motor control based on event-based hybrid neuromorphic-procedural system," in *2007 IEEE International Symposium on Circuits and Systems*, pp. 845–848, May 2007. ISSN: 2158-1525.

[256] H. Rebecq, R. Ranftl, V. Koltun, and D. Scaramuzza, "High Speed and High Dynamic Range Video with an Event Camera," *IEEE transactions on pattern analysis and machine intelligence*, vol. 43, pp. 1964–1980, June 2021.

[257] P. Lichtsteiner, C. Posch, and T. Delbruck, “A 128$\times$ 128 120 dB 15 μs Latency Asynchronous Temporal Contrast Vision Sensor,” *IEEE Journal of Solid-State Circuits*, vol. 43, pp. 566–576, Feb. 2008. Conference Name: IEEE Journal of Solid-State Circuits.

[258] S. Barua, Y. Miyatani, and A. Veeraraghavan, “Direct face detection and video reconstruction from event cameras,” in *2016 IEEE Winter Conference on Applications of Computer Vision (WACV)*, (Lake Placid, NY, USA), pp. 1–9, IEEE, Mar. 2016.

[259] J. Yang, W. Zhao, Y. Han, C. Ji, B. Jiang, Z. Zheng, and H. Song, “Aircraft tracking based on fully conventional network and Kalman filter,” *IET Image Processing*, vol. 13, no. 8, pp. 1259–1265, 2019.

[260] H. Liu, D. P. Moeys, G. Das, D. Neil, S.-c. Liu, and T. Delbrück, “Combined frame- and event-based detection and tracking,” in *2016 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 2511–2514, 2016.

[261] B. Ramesh, S. Zhang, H. Yang, A. Ussa, M. Ong, G. Orchard, and C. Xiang, “e-TLD : Event-based Framework for Dynamic Object Tracking,” *arXiv preprint arXiv:2009.00855*, pp. 1–11, 2020. arXiv: 2009.00855v1.

[262] S. Paul, T. Majumder, C. Augustine, A. F. Malavasi, S. Usirikayala, R. Kumar, J. Kollikunnel, S. Chhabra, S. Yada, M. L. Barajas, C. Ornelas, D. Lake, M. M. Khellah, J. Tschanz, and V. De, “A 0.05pJ/Pixel 70fps FHD 1Meps Event-Driven Visual Data Processing Unit,” in *2020 IEEE Symposium on VLSI Circuits*, (Honolulu, HI, USA), pp. 1–2, IEEE, June 2020.

[263] X. Lagorce, C. Meyer, S. H. Ieng, D. Filliat, and R. Benosman, “Asynchronous Event-Based Multikernel Algorithm for High-Speed Visual Features Tracking,” *IEEE Transactions on Neural Networks and Learning Systems*, vol. 26, no. 8, pp. 1710–1720, 2015. Publisher: IEEE.

[264] S. K. Bose and A. Basu, “A 389TOPS/W, 1262fps at 1Meps Region Proposal Integrated Circuit for Neuromorphic Vision Sensors in 65nm CMOS,” in *2021 IEEE Asian Solid-State Circuits Conference (A-SSCC)*, (Busan, Korea, Republic of), pp. 1–3, IEEE, Nov. 2021.

[265] S. K. Bose, D. Singla, and A. Basu, “A 51.3 TOPS/W, 134.4 GOPS In-memory Binary Image Filtering in 65nm CMOS,” *IEEE Journal of Solid-State Circuits*, vol. 57, pp. 323–335, Jan. 2022.

[266] S. Gupta, M. Imani, J. Sim, A. Huang, F. Wu, J. Kang, Y. Kim, and T. S. Rosing, “COSMO: Computing with Stochastic Numbers in Memory,” *ACM Journal on Emerging Technologies in Computing Systems*, vol. 18, pp. 1–25, Apr. 2022.

[267] J. Yue, X. Feng, Y. He, Y. Huang, Y. Wang, Z. Yuan, M. Zhan, J. Liu, J.-W. Su, Y.-L. Chung, P.-C. Wu, L.-Y. Hung, M.-F. Chang, N. Sun, X. Li, H. Yang, and Y. Liu, "A 2.75-to-75.9TOPS/W Computing-in-Memory NN Processor Supporting Set-Associate Block-Wise Zero Skipping and Ping-Pong CIM with Simultaneous Computation and Weight Updating," in *2021 IEEE International Solid- State Circuits Conference (ISSCC)*, (San Francisco, CA, USA), pp. 238–240, IEEE, Feb. 2021.

[268] H. Jia, M. Ozatay, Y. Tang, H. Valavi, R. Pathak, J. Lee, and N. Verma, "Scalable and Programmable Neural Network Inference Accelerator Based on In-Memory Computing," *IEEE Journal of Solid-State Circuits*, vol. 57, pp. 198–211, Jan. 2022.

[269] J. Fromm, M. Cowan, M. Philipose, L. Ceze, and S. Patel, "Riptide: Fast End-to-End Binarized Neural Networks," *Proceedings of Machine Learning and Systems*, vol. 2, pp. 379–389, 2020. arXiv: 1604.03058v5.

ProQuest Number: 30312902

INFORMATION TO ALL USERS
The quality and completeness of this reproduction is dependent on the quality
and completeness of the copy made available to ProQuest.

Distributed by ProQuest LLC (2023).
Copyright of the Dissertation is held by the Author unless otherwise noted.

This work may be used in accordance with the terms of the Creative Commons license
or other rights statement, as indicated in the copyright statement or in the metadata
associated with this work. Unless otherwise specified in the copyright statement
or the metadata, all rights are reserved by the copyright holder.

This work is protected against unauthorized copying under Title 17,
United States Code and other applicable copyright laws.

Microform Edition where available © ProQuest LLC. No reproduction or digitization
of the Microform Edition is authorized without permission of ProQuest LLC.

ProQuest LLC
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 - 1346 USA