**Title**
Learned Approximate Computing for Machine Learning



**Author**
Li, Tianmu

**Publication Date**
2023

Peer reviewed|Thesis/dissertation

UNIVERSITY OF CALIFORNIA

Los Angeles

# Learned Approximate Computing for Machine Learning

A dissertation submitted in partial satisfaction

of the requirements for the degree

Doctor of Philosophy in Electrical and Computer Engineering

by

Tianmu Li

2023

ABSTRACT OF THE DISSERTATION

Learned Approximate Computing for Machine Learning

by

Tianmu Li
Doctor of Philosophy in Electrical and Computer Engineering
University of California, Los Angeles, 2023
Professor Puneet Gupta, Chair

Machine learning using deep neural networks is growing in popularity and is demanding increasing computation requirements at the same time. Approximate computing is a promising approach that trades accuracy for performance, and stochastic computing is an especially interesting approach that preserves the compute units of single-bit computation while allowing adjustable compute precision. This dissertation centers around enabling and improving stochastic computing for neural networks, while also discussing works that lead up to stochastic computing and how the techniques developed for stochastic computing are applied to other approximate computing methods and applications other than deep neural networks. We start with 3pxnet, which combines extreme quantization with model pruning. While 3pxnet achieves extremely compact models, it demonstrates limits of binarization, including the inability to scale to higher precision levels and performance bottlenecks from accumulation. This leads us to stochastic computing, which performs single-gate multiplications and additions on probabilistic bit streams. The initial SC neural network implementation in ACOUSTIC aims at maximizing SC performance benefits while achieving usable accuracy. This is achieved through design choices in stream representation, performance optimizations

using pooling layers, and training modifications to make single-gate accumulation possible. The subsequent work in GEO improves the stream generation and computation aspects of stochastic computing and reduces the accuracy gap between stochastic computing and fixed-point computing. The accumulation part of SC is further optimized in REX-SC, which allows efficient modeling of SC accumulation during training. During these iterations of the SC algorithm, we developed efficient training pipelines that target various aspects of training for approximate computing. Both forward and backward passes of training are optimized, which allows us to demonstrate model convergence results using SC and other approximate computing methods with limited hardware resources. Finally, we apply the training concept to other applications. In LAC, we show that an almost arbitrary parameterized application can be trained to perform well with approximate computing. At the same time, we can search for the optimal hardware configuration using NAS techniques.

The dissertation of Tianmu Li is approved.

Yizhou Sun

Nader Sehatbakhsh

Sudhakar Pamarti

Puneet Gupta, Committee Chair

University of California, Los Angeles

2023

*To my parents for their support.*

TABLE OF CONTENTS

**1 Introduction** . . . . . **1**

1.1 Computation in deep neural networks . . . . . 1

1.2 Computation with reduced accuracy . . . . . 4

**2 Sparse Binarized Processing** . . . . . **9**

2.1 Introduction . . . . . 9

2.1.1 A case for sparse XNOR networks . . . . . 10

2.2 Related Work . . . . . 12

2.2.1 Binarized neural networks . . . . . 12

2.2.2 Weight pruning . . . . . 13

2.2.3 Sparse binary networks . . . . . 14

2.2.4 Machine learning on embedded systems . . . . . 14

2.3 The 3PXNet Approach . . . . . 15

2.3.1 XNOR networks . . . . . 15

2.3.2 Challenges in pruning XNOR networks . . . . . 15

2.3.3 Pruning a packed XNOR network . . . . . 17

2.3.4 Training 3PXNet . . . . . 19

2.4 Experimental Setup . . . . . 21

2.4.1 Platforms . . . . . 21

2.4.2 Benchmarks . . . . . 21

2.4.3 Baseline . . . . . 22

2.5 Results and Discussion . . . . . 23

2.5.1 Accuracy & model size . . . 23
2.5.2 Performance & energy . . . 26
2.6 Conclusion . . . 27

**3 Making Stochastic Computing Work for Deep Learning . . . 29**
3.1 Stochastic Computing Primer . . . 31
3.1.1 Error behavior . . . 31
3.1.2 Stream generation . . . 32
3.1.3 Computation . . . 33
3.1.4 Storage . . . 38
3.1.5 Non-linear computation . . . 38
3.2 Stochastic Computing Baseline Implementation . . . 38
3.2.1 Split-unipolar stream representation . . . 38
3.2.2 Computation skipping for average pooling . . . 40
3.2.3 Making OR accumulation practical . . . 40
3.2.4 Modeling the error of stochastic computing . . . 41
3.3 Conclusion . . . 43

**4 Improving SC Generation and Execution . . . 44**
4.1 Co-optimized Shared Generation and Training . . . 45
4.2 Partial Binary Accumulation . . . 46
4.3 Results . . . 49
4.3.1 Evaluation methodology . . . 49
4.3.2 GEO accuracy comparisons . . . 50

4.3.3 Performance Impact of GEO Enhancements . . . 51
4.4 Conclusion . . . 52

**5 Optimizing SC accumulation . . . 53**
5.1 Motivation . . . 55
5.1.1 SC accumulation primer . . . 56
5.1.2 Precision . . . 56
5.1.3 Efficiency . . . 58
5.2 Range-extended OR Accumulation . . . 60
5.2.1 Increasing the accuracy of OR accumulation . . . 60
5.2.2 Efficient OR_n implementation . . . 66
5.2.3 Efficient training for OR_n accumulation . . . 68
5.3 Results . . . 75
5.3.1 Evaluation . . . 75
5.3.2 Accuracy improvements . . . 76
5.3.3 Training speed improvements . . . 78
5.3.4 Performance . . . 80
5.3.5 OR_n for non-trained applications . . . 82
5.4 Conclusion . . . 83

**6 Solving the Training Issue . . . 84**
6.1 Speeding up SC simulation . . . 86
6.1.1 Stream packing . . . 87
6.1.2 Computation optimization . . . 92

6.1.3 Generation optimization . . . 100
6.2 Training Neural Networks for Execution on Approximate Computing Hardware 100
6.2.1 Activation proxy function . . . 100
6.2.2 Error injection training . . . 103
6.2.3 Fine tuning . . . 107
6.2.4 Gradient checkpointing . . . 108
6.2.5 Results . . . 109
6.3 Conclusion . . . 111

**7 Learned Approximate Computing . . . 113**
7.1 Introduction . . . 113
7.2 Fixed Hardware LAC . . . 115
7.2.1 Training applications for a fixed hardware . . . 116
7.3 Evaluation on Fixed Hardware . . . 117
7.3.1 Approximate hardware . . . 117
7.3.2 Applications . . . 118
7.3.3 Dataset . . . 120
7.3.4 Optimization solvers . . . 120
7.3.5 Results . . . 122
7.4 Trained Hardware LAC . . . 123
7.5 Evaluation on Trained Hardware . . . 126
7.6 Conclusion . . . 128

**8 Conclusion . . . 129**

8.1 Validation of SC-based Neural Networks on More Models and Datasets . . . 131
8.2 Efficient Model Transfer for Stochastic Computing . . . 132
8.3 Enabling SC for Large Language Models . . . 133
8.4 SC for Homomorphic Encryption . . . 133
8.5 Specialized SC Architectures for Emerging Compute Paradigms . . . 134
8.5.1 Range-adjustable OR_n accumulation . . . 134
8.5.2 Delta-sigma SC . . . 135

**References . . . 138**

LIST OF FIGURES

2.1 Storage requirements for Dense, NP, and NPP XNOR "small" MNIST MLP for varying levels of sparsity. . . . . 16

2.2 Pruning with packing constraint of 4 bits a) without permutation, b) with permutation . . . . . 18

2.3 Comparison of training results with and without permutation for different sparsities. 18

2.4 Accuracy vs. Memory tradeoff compared to eBNN and dense XNOR. . . . . 23

2.5 Accuracy comparison between sparse 8-bit network and 3PXNet, for MNIST (a) and CIFAR-10 (b). . . . . 25

3.1 Relative error resulting from (a) quantization and (b) randomness of stochastic computing. . . . . 34

3.2 Comparison of MUX- and OR-based adders' error with respect to accurate addition. 37

3.3 Accuracy comparison between MUX and OR a) and comparison of approximation methods for OR accumulation b). . . . . 42

4.1 Accuracy vs. sharing for TRNG and LFSR-based random number generation. . 47

4.2 Area comparison for different hardware implementations of SC-based MAC units for different kernel sizes and different levels of partial binary accumulation. . . . 48

4.3 Area, energy, and latency comparison between baseline and different GEO configurations (normalized to Base-128,128). The first bar in every group is for area while the second is for energy. . . . . 51

5.1 (a) best and (b) worst case for partial binary accumulation. . . . . 54

5.2 Area (a), critical path (b), and area-delay product (ADP) (c) for different dot product implementations. For SC implementations, ADP is calculated after multiplying the stream length, which is assumed to be 32 for each split-unipolar part. . . . . 59

5.3 Overview comparison between (a) OR and (b) OR_n. . . . . 61

5.4 (a) Single OR_n gate and (b) cascaded OR_n gates. . . . . 61

5.5 OR_2 circuit implementation. Similar to Tab. 5.2, a, b, c, and d are the four input bits, and e and f are the two output bits. . . . . 67

5.6 ADP comparison between different OR_2 truth table implementations. . . . . 67

5.7 (a) Parallel and (b) saturating adder implementation of OR_3. . . . . 68

5.8 TinyConv accuracy of with fixed-point computation using different intermediate precision. . . . . 69

5.9 Modeling performance of (a) OR_2 and (b) OR_3. . . . . 70

5.10 Modeling performance comparison of OR_2 and 2-way partial binary accumulation. pb_r and pb_s use the same inputs and only the order of computation is changed, such that for pb_r partial sums are roughly equal, and for pb_s they are very different. . . . . 71

5.11 Difference between outputs from stream computation and normal computation+activation. 72

5.12 Comparison between (a) vanilla STE [Ben13], (b) stream with OR_n activation and (c) a+e training step. . . . . 73

5.13 Accuracy comparison for (a) TinyConv and (b) VGG-16 on CIFAR-10, (c) Resnet-18, and (d) Resnet-34 on ImageNet. 16-bit SC-Bin only has an accuracy result for TinyConv as it does not converge for other larger models. . . . . 75

5.14 Convergence behavior of stream only and a+e. The drop in accuracy at epoch 31 for a+e is when switching to using streams for the forward pass. . . . . 79

5.15 Comparison of multiplexer- and OR_n-based adders' root mean squared error (RMSE) with respect to normal addition. We use sums of 1000 positive inputs with outputs having an average of 1 and a variance of 0.5. . . . . 82

6.1 Comparison between (a) SC AND-OR dot product and (b) an equivalent implementation using single-bit multiply-add operations. . . . . 88

6.2 Comparison between stream generation for packing in (a) bitstream and (b) input channel dimension for a single thread. . . . . 90

6.3 (a) Naive memory layout, (b) strided memory layout used, and (c) padded memory layout for an 8x32(32-bit) array. The colored cells are for data storage, and the rest are either index (first row and column) or unfilled storage in (c). The numbers in the cell have slightly different meanings. In (a), the number represents both the column index in a column-major matrix and the bank index, as the number of columns matches the number of banks. In (b), the number represents the column index, and the bank index matches the column number. In (c), the number presents the bank index, and the column index matches the column number. . . . . 94

6.4 Activation modeling behavior of unipolar and bipolar (a) stochastic and (b) analog computation. The bipolar versions performs subtraction between two unipolar (positive and negative) inputs to achieve the full range. For analog computation, the ADC saturation is modeled as a clamp at 2 in this example, and other effects (e.g. size of accumulation) are not considered. . . . . 102

6.5 Difference between outputs from stream computation and normal computation+activation. 106

6.6 Convergence behavior of TinyConv with and without error injection using (a) stochastic computing and (b) approximate multiplication. Stochastic computing and approximate multiplication models are trained from scratch for 100 epochs and the first 20 epochs and the last few epochs are shown. "Error X" means training with error injection with X epochs of fine-tuning with an accurate model. "No Error X" means training without error injection with X epochs of fine-tuning. "Model" means accurate modeling throughout training. . . . . 108

6.7 End-to-end runtime improvements. Time is measured in hours required to converge. The "Without Improvements" results are estimated for models that are impossible to train without the improvements. . . . . 112

7.1 Difference between traditional and LAC setups. LAC focuses on optimizing the application kernels rather than optimizing the hardware approximations. . . . . 115

7.2 Overview of LAC for fixed hardware. . . . . 116

7.3 Quality improvements of (a) Gaussian blur, (b) edge detection, (c) image sharpening, (d) JPEG compression using DCT, (e) DFT, and (f) Inversek2j. . . . . 121

7.4 Quality improvements of (a) Gaussian blur, (b) edge detection, and (c)image sharpening. Figures on the right use Pareto optimal multipliers with the highest SSIM before optimization. . . . . 121

7.5 Overview of LAC for trained hardware. . . . . 123

7.6 Illustration of the binarized gate used for choosing between different hardware outputs. . . . . 123

7.7 LAC search results of (a) Gaussian blur, (b) edge detection, (c) image sharpening, (d) JPEG compression using DCT, (e) DFT, and (f) Inversek2j. . . . . 126

7.8 LAC performance-centric search results of (a) Gaussian blur, (b) edge detection, (c) image sharpening, (d) JPEG compression using DCT, (e) DFT, and (f) Inversek2j. . . . . 127

8.1 Accumulation demonstration of OR-based SCIM accumulation . . . . . 135

8.2 Accumulation demonstration of SD SCIM accumulation . . . . . 136

8.3 Accuracy of SC with sigma-delta ADC compared to OR SC and SC with binary accumulation for (a) TinyConv on CIFAR-10 and (b) Resnet-18 on ImageNet. . 137

LIST OF TABLES

2.1 Weight storage requirements of different networks depending on precision. . . . 11

2.2 Hardware platforms used for the runtime experiments. Only the *Small* platform has a DSP extension with hardware multiply-accumulate unit. All three microcontrollers are from the ST Nucleo family. . . . . 21

2.3 Benchmark models and datasets . . . . . 22

2.4 Accuracy and network size (KB, in brackets) comparison. . . . . 23

2.5 Runtime (ms) and energy (mJ, in brackets) for MNIST networks. A dash indicates a given model could not fit in memory. . . . . 27

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. . . . . 27

3.1 Validation accuracy of a 4-layer CNN[LSC18] for SC hardware with different training methods. . . . . 43

4.1 Accuracy comparison with fixed-point, other SC implementations, and so on. . . 50

5.1 Summary of various SC addition methods and works using them. . . . . 55

5.2 Comparison between a full truth table and a sum transfer function. This is one possible truth table of a two-input OR_2 gate that corresponds to Candidate 6 in Tab. 5.3. This particular truth table sets the two output bits to $e = a + c + bd$ and $f = b + d + ac$ respectively, assuming $a$, $b$, $c$, and $d$ are the four input bits, and $e$ and $f$ are the two output bits. . . . . 63

5.3 Candidate transfer functions between input and output sums for 2-input OR_2. 64

5.4 Candidate transfer functions between input and output sums for 3-input OR_2. 65

5.5 TinyConv Neural network accuracy of Candidate 1, 4, and 6 for OR_2. Models are trained using 32-bit streams on CIFAR-10. . . . . 66

5.6 LFSR seed sharing scheme demonstration. . . . . 76

5.7 Array size, compute area and power, clock period, inference latency, energy, and energy improvement compared to 6-bit fixed-point, for different models and datasets with different types of SC accumulation. . . . . 77

5.8 Iteration time comparisons between OR_n and partial binary accumulation. Time is measured in s/256 images on an RTX 3090. The numbers before "PB" are the sizes of OR accumulation, with smaller numbers leading to more binary accumulations. . . . . 78

5.9 End-to-end training time comparison in hours between OR_n and partial binary accumulation. Results for OR_n and PB are estimated as they take prohibitively long to train. . . . . 80

6.1 Operator and data format comparison between bitstream and input channel packing 89

6.2 Accuracy benefits of using activation functions. . . . . 101

6.3 Activation functions for stochastic computing and analog computing. $x$ is the output of a layer before activation. $x_{pos}$ is the output of positive weights and $x_{neg}$ is the output of negative weights. Inputs are assumed to be non-negative due to ReLU activation. . . . . 103

6.4 Relative multiplication and addition cost. FP32 multiplication and addition are used as the baseline. The number of operations in C++ is used as the cost value. 104

6.5 Accuracy impact of modeling approximate computation. "With Model" models the approximate computation method accurately in the forward pass. . . . . 104

6.6 Accuracy impact of error injection training. . . . . 107

6.7 Training runtime and memory requirement of stochastic computing without and without gradient checkpointing. We use the maximum batch size achievable which is a power of 2. . . . . 109

6.8 Runtime impact of error injection training. Shown is the time (seconds) required per epoch. Runtimes are measured on an RTX 3090 using TF32 precision and batch size=256. SC and approximate multiplication are slower due to the need to split positive and negative computations discussed in Sec. 6.2.1. Fine-tuning runtime is the same as the runtime with accurate model (the "With Model" column). . . . . 110

6.9 Epochs used for training. Methods with higher accuracy require fewer epochs of fine-tuning. . . . . 110

6.10 Top-1 accuracy of Resnet-18 on ImageNet. . . . . 111

7.1 Multiplier summary. Performance numbers are normalized to 16-bit multipliers. 118

7.2 Application summary . . . . . 118

# ACKNOWLEDGMENTS

This dissertation is the culmination of years of work, and would not have been possible without the inspiration and guidance from a lot of people. I would like to use this as an opportunity to express my gratitude.

First, I would like to thank my advisor Prof. Puneet Gupta. None of the work in this dissertation would have been possible without his help. I am very grateful for him giving me this opportunity to study and perform research in this lab and also for his constant support and guidance throughout the many years I spent in this lab.

I would like to especially thank my lab mate, Wojciech Romaszkan. He is a collaborator for most of the works listed in this dissertation. His contributions to the performance and architectural analysis have been crucial to these works. He has also been extremely helpful throughout my life in the lab and in the various other projects that we collaborated on.

I would like to thank my undergrad mentor, Shaodi Wang. He was an alumnus of this lab and was my mentor when I worked as an undergrad. It was under his guidance that I learned the concepts of stochastic computing. The exploratory research I performed as an undergrad eventually formed the basis of this dissertation, and our work on stochastic computing using magnetic tunnel junctions formed the basis of a proposal. That proposal eventually lead to the project that funded my Ph.D. career.

I would like to thank my mentors and colleagues at ARM and Intel, where I joined as an intern. While the projects I worked on during my internships had little in common with the overall scheme of this dissertation, they both involved software optimizations to improve deep learning performance. It's through these internship experiences that I learned the value of low-level optimizations. These optimizations never became the centerpiece of my research, but they are an enabler of a lot of the experiments.

I would like to thank all my other labmates that accompanied me throughout this process. They include Weiche Wang, Saptadeep Pal, Irina Alam, Shurui Li, Alexander Graening,

and Rhesa M. Ramadhan (ordered by their (prospective) graduation date). NanoCAD lab really emphasizes in-person communications, and I enjoyed working with all my labmates throughout this process. We worked in the same room, collaborated on various projects, and ate out now and then. The conversations we had throughout the years have been extremely inspirational to my research, and the various activities we participated in added variations to my otherwise straightforward Ph.D. career.

Finally, I would like to thank my parents for their continued support. It's always reassuring to know that I always have a place to return to, and they backed me up through all the difficulties I had.

# VITA

2013–2017 B.S. (Electrical Engineering), UCLA.

2017–present Graduate Student Researcher, UCLA.

# PUBLICATIONS

Wojciech Romaszkan, **Tianmu Li**, and Puneet Gupta. "3PXNet: Pruned-Permuted-Packed XNOR Networks for Edge Machine Learning." *ACM Transactions on Embedded Computing Systems (TECS)*, November 2019.

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)*, 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)*, pp. 1–6, 2021.

Vaibhav Gupta, **Tianmu Li**, and Puneet Gupta, "LAC: Learned Approximate Computing." In *IEEE/ACM Design, Automation and Test in Europe*, p. 4, March 2022.

**Tianmu Li**, Shurui Li, and Puneet Gupta. "Training Neural Networks for Execution on Approximate Hardware." In *International Research Symposium on Tiny Machine Learning (tinyML)*, p. 6, March 2023.

**Tianmu Li**, Wojciech Romaszkan, Sudhakar Pamarti, and Puneet Gupta. "REX-SC: Range-Extended Stochastic Computing Accumulation for Neural Network Acceleration." *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, June 2023

# CHAPTER 1

# Introduction

Machine learning has seen massive developments in recent years, and state-of-the-art models have been pushing the boundary of what can be modeled using neural networks. However, model capacity has greatly outpaced the development of hardware. Popular large language models (LLMs) require a large cluster with thousands to tens of thousands of GPUs to train in a reasonable time [HBM22]. Even performing inference of the trained model can be prohibitively expensive [PA]. As a result, a lot of machine learning inference tasks run in the cloud. However, running everything in the cloud creates multiple issues. Network connectivity is a concern, and the strict power budget of resource-constrained devices can prevent the usage of relatively expensive network connections. Transmitting training and inference data over the network also creates privacy and security concerns. As a result, there is always interest in performing local inference on resource-constrained devices.

## 1.1 Computation in deep neural networks

At the core of today's high-performance machine learning models are deep neural networks. Neural networks consist of multiple layers connected in different fashions, and the way connections happen within and between layers defines the type of neural network. Multi-layer perceptrons connect layers sequentially, with an all-to-all connection within each layer. Reduced connections within layers generate convolutional neural networks (CNN) [LBB98]. Cyclic connections between layers generate recurrent neural networks (RNNs) [RM87]. Connections with themselves in a layer generate attention layers that form the backbone of

transformers [VSP17]. While neural networks come in numerous variants with the number of variants still rapidly increasing, the core computation is defined by the connections within a layer, which are made up of multiply-accumulate operations, or dot products. To improve computation efficiency, most of the computations in neural networks contain dot products of the same size, which can be mapped to matrix multiplications. Apart from linear matrix computations, neural networks also have non-linear computations between layers, also known as activation functions. This non-linear computation forms the basis of the universal approximation property of neural networks [HSW89]. Despite their importance, they are typically made up of element-wise operations where the same operation is applied to all elements (usually) equally. Compared to the dot product and matrix multiplication computations, activation functions are typically cheaper to implement, and can often be fused with dot product kernels in libraries like cuDNN and oneDNN to further reduce their impact on performance. As a result, dot products and matrix multiplications remain the dominant computation in today's neural networks.

Computations in neural networks come in two variants, training, and inference. Training refers to modifying the parameters of the neural network so that the output of the model is close to the desired output. If the desired output is manually labeled, this process is called supervised learning. An example is the ImageNet challenge [RDS15], where the model needs to create a label for a given image. During training, the label is provided as the desired output. If the desired output is not labeled, this process is called unsupervised learning. An example of unsupervised learning is the pre-training phase of language models. The popular Bidirectional Encoder Representations for Transformers (BERT) [DCL18] is pre-trained on English Wikipedia for instance. During pre-training, the model needs to predict the content of a randomly masked word in a sentence or predict whether a sentence is the correct next sentence. Unsupervised learning is sometimes not sufficient to generate a model suitable for a given task, in which case the model needs to be fine-tuned on the specific task afterward. Regardless of whether supervised or not, the training of neural networks

can be further broken down into two passes, the forward pass and the backward pass. The forward pass computes the final output and intermediate outputs of the model for a given (minibatch of) input. The final output is compared with the desired output and generates a loss value using a loss function. During the backward pass, the loss value is used to calculate the gradient of different weight values through backpropagation [RHW86]. The backpropagation step can be seen as the inverse of the computation in the forward pass. For the dot product/matrix multiplication part of the computation, the backpropagation step is separated into backward_input and backward_weight steps. [1] In the backward_input step, the gradient with respect to the input values is computed using the output gradient value and weight values. The input gradient will then become the output gradient of the previous layer. In the backward_weight step, the gradient with respect to the weight values is computed using the output gradient value and output value. As a result, the intermediate output values of layers need to be stored for the backward pass (there are methods to trade this memory requirement for more computation, which will be discussed in Chap. 6). After backpropagation through all layers, the weight gradients are used to update the weight values using an optimizer. Optimizer options range from simple ones like stochastic gradient descent (SGD), or adaptive ones that modify the per-weight update rate like Adam [KB14].

After training a neural network, it can then be used for inference, which refers to running the forward pass for a given prediction task.[2] During inference, the weights are typically fixed and don't need to be updated. As a result, the intermediate output of layers can be discarded during inference. Whereas training performs three matrix multiplications, one in the forward pass and two in the backward pass, inference only performs one. This makes neural network inference generally cheaper than training. Despite the reduced cost for a

[1] Backpropagation through activation layers only have backward_input step as there are no weights. Backpropagation through matrix multiplication of two input matrices (self-attention) has two backward_input steps, as both operands of the matrix multiplication are inputs.

[2] Generation (as is now popular with likes of Stable Diffusion [RBL21] for image generation and ChatGPT [Ope23] for text generation) can be viewed as the prediction of the next pixel or the prediction of the next word.

single input, the cost of inference can become dominant in the long run. Once trained, a single model may be used for inference for a lot more inputs than being used during training. This factor makes neural network inference acceleration important despite the apparent low cost compared to training for the same number of inputs.

Between the two variants of neural network computation, training generally requires higher-precision computation. As the range of weight and input values of a layer can change dramatically during training, floating-point number formats are generally preferred over fixed-point numbers. Inference, on the other hand, has lower precision requirements in general. Since the weight and (to some extent) input distributions are fixed during inference, fixed-point quantization generally performs well enough during inference once the range of inputs and weights is correctly adjusted.

## 1.2 Computation with reduced accuracy

As neural network inference can be the dominant computation and requires less precision than training, multiple computation methods have been proposed to improve the efficiency of neural network inference at the cost of reduced accuracy for single computation.

One line of work continues from the previously-mentioned fixed-point quantization. Compared to computation using 32-bit floating-point numbers, 8-bit quantized weights and computation reduce both computation and memory costs during inference. In terms of accuracy, 8-bit fixed-point numbers have been shown to lose only a little accuracy when quantizing from a floating-point model without training. Quantization without training is known as post-training quantization (PTQ) and allows easy deployment of floating-point models with fixed-point computation [Mig17]. For cases where training is allowed, quantization-aware training (QAT) can further improve accuracy [JKC18, ZWN16]. In the most common form, QAT performs fake quantization during part of or entire training. In a fake quantization setup, a copy of weight values is kept at high precision, which is also known as master weights.

In the forward pass, the master weights are quantized to compute layer outputs. In the backward pass, these outputs are used to compute the weight gradients using high-precision computation. The gradients are then used to update the high-precision master weight values. This setup is known as a straight-through estimator (STE) [Ben13]. Training-aware quantization further reduces the accuracy loss induced by quantization, and a lot of models lose very little to no accuracy with 8-bit quantization-aware training. With the success of 8-bit quantization, there has been continued work to push the precision further down. 4-bit quantization on weights has been shown to be sufficient for various models [BNS19]. In the extreme case, 1-bit quantization was proposed to limit both storage and computation to single bits [CHS16] at the cost of some accuracy loss.

Another line of work focuses on dropping unimportant computations in neural networks altogether, also known as pruning. As neural network training uses a first-order optimization algorithm, training convergence is typically slower and noisier compared to more traditional optimization algorithms like Newton's method which require higher-order information. As a result, there are observations that a lot of the weights in the trained neural networks are redundant, and can be effectively dropped without significantly impacting the final accuracy [HPT15]. Dropping weights in a neural network can theoretically save both the weight storage cost and the associated computation cost.

The last line of work tries to use approximate computing methods to further reduce the cost of computation. Compared to quantization which tries to perform the same operations but at lower precision or pruning which tries to remove unimportant computation, approximate computing deliberately introduces errors or uncertainties into the computation. Some examples of approximate computing include approximate arithmetic, analog computation, and stochastic computation. Approximate arithmetic tries to simplify the digital arithmetic circuits, typically multipliers, even if such simplification generates errors for certain input computations. With careful selection of the places to simplify, approximate computing can further reduce the area and power of arithmetic operations with only a small loss to model

accuracy [GSG22]. Analog computing tries to perform computations using circuits not originally designed for computation, typically memory [YJH21]. Performing computation in memory arrays offers multiple benefits. On the one hand, this saves memory movement costs, as weights are stored in the memory array and don't need to be read out. On the other hand, novel memory technologies like magnetic memory (MRAM) [POZ18] and resistive memory (RRAM) [CLX16] improve computation efficiency compared to fixed-point and floating-point computation.

Stochastic computing (SC) is another one of the approximate computing methods. SC performs computation using randomized bitstreams and is shown to be promising for neural network acceleration [SNL17]. With the stream representation, stochastic computing enables multiplications and additions using single gates. This setup significantly improves the computation density. The improved density in turn increases input value reuse during computation, which reduces memory movement requirements [RLM20]. However, these benefits don't come for free. As is obvious from the name, stochastic computing is random. Both the generation and computation of bit streams create errors, which hampers the accuracy of stochastic computing.

This thesis mainly focuses on improving the efficiency of neural network inference using stochastic computing. Apart from the implementation of SC for neural networks, this thesis will also discuss the motivations for using SC in the first place, and also discuss the application of the training techniques for SC-based neural networks on other approximate computation methods and other applications.

The thesis will be organized as follows:

- Chapter 2 will discuss 3pxnet: packed, pruned, and permuted xnor-net, which aims to achieve maximal compression of a model by combining 1-bit quantization and pruning. By performing multiple optimizations including bit packing and input channel permutation, we demonstrate further memory reduction and performance gains over model

binarization and pruning alone.

- Chapter 3 will discuss my contributions in ACOUSTIC: Accelerating Convolutional Neural Networks through Or-Unipolar Skipped Stochastic Computing. It will start with how we arrived at stochastic computing from the initial work on model binarization and pruning. Following a brief introduction to SC will be the initial implementation of an SC-based neural network, which serves as a basis for further optimizations.

- Chapter 4 will discuss my work on GEO: Generation and Execution Optimized Stochastic Computing Accelerator for Neural Networks. It includes optimizations to the generation and execution components of stochastic computing. These optimizations improve the accuracy-performance tradeoff of stochastic computing.

- Chapter 5 will discuss my work on REX-SC: range-extended stochastic computing for neural network execution. While the generation optimizations of GEO are mostly sufficient, the execution optimizations introduce multiple complications. REX-SC introduces OR-n as an alternative to execution optimization in GEO. OR-n enables more efficient modeling during training, which allows us to demonstrate the benefits of SC on larger models and more complicated datasets.

- Chapter 6 will discuss my efforts to improve the training performance for neural networks using stochastic computing and other approximate computing methods in general. This chapter discusses the difficulties faced during training along with solutions to each one of them. The introduced optimizations act as enablers for the models used in Chapter 4.

- Chapter 7 will discuss the application of the training idea to other applications and other approximate computing methods. In LAC; learned approximate computing search, we allow optimization of an almost arbitrary parameterizable application for a particular approximate hardware configuration. For a fixed approximate hardware

configuration, LAC improves application performance and allows hardware reuse. For cases where the hardware can also be changed, we enable an automatic search of the hardware configuration that finds the hardware with the best accuracy-performance tradeoff after training.

- Chapter 8 will provide a conclusion of the thesis along with some promising directions for future research.

# CHAPTER 2

# Sparse Binarized Processing [1]

## 2.1 Introduction

With the increasing need for intelligence in Internet-of-things devices, there is a growing interest in deploying neural networks on mobile and edge devices. However, state-of-the-art deep learning models have sizes in tens or hundreds of megabytes and require millions of multiply-accumulate operations, making them impossible to use on heavily resource-constrained platforms.

To cope with this, various model compression schemes have been developed in recent years, chief among them, quantization and pruning [CWZ17]. Multiple works have shown that decreasing the precision of underlying computation through quantization does not affect accuracy, while significantly improving storage and runtime [VSM11, CCR13, GAG15, ZWN16, JAM17, JBB17]. In the most extreme case of precision reduction, binarization can be used without a significant drop in accuracy [CSM15, CHS16, KS16, ROR16, CHS16].

Exploiting redundancy through sparsity has been studied since the advent of neural networks [LBB98, HSW93], and recent work [HPT15] has shown over 10x compression on popular network models with the same accuracy. Although model compression through pruning significantly reduces the required computational complexity, it is hard to efficiently

[1] **This work is performed in collaboration with Wojciech Romaszkan and Puneet Gupta. This chapter contains material previously used in [RLG19]. My contributions to this work include training the models, including the fixed-point baseline and dense and sparse binarized networks, and developing the methodologies around the training and pruning components.**

exploit, especially on highly-parallel hardware [YLP17]. Combining binarization and pruning is the next logical step when pushing the limit of model compression [YSN18]. Unfortunately, naively induced sparsity makes exploiting binarization-enabled parallelism prohibitively costly. To achieve actual performance gains over dense binarized implementation, a form of structured pruning needs to be employed [LZP17]. However, this enforced structure might in turn constrain pruning flexibility and negatively affect the accuracy of the network, which in itself is undesirable.

### 2.1.1 A case for sparse XNOR networks

Table 2.2 shows available memory in some of the common embedded microcontroller platforms. This is usually limited to a few hundred kilobytes at most. Such severe resource constraints make the implementation of reasonable deep learning networks on these platforms very challenging. For example consider a few network models shown in Table 2.1. Most floating point and even 8-bit fixed point implementations are 10-1000X off from where they need to be for these platforms.

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 2.1). While this in itself is impressive, it might not be enough to use common models on typical embedded development platforms. Further compression is, therefore, necessary to use large models on those devices, or in the case of the most memory-constrained ones, make it feasible to deploy them at all.

In this paper, we propose a Pruned-Permuted-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

Table 2.1: Weight storage requirements of different networks depending on precision.

| Network | Weight Memory [MB] | | |
|---|---|---|---|
| | F32 | FP8 | XNOR |
| ILSVRC VGG-D [SZ14] | 553.4 | 138.3 | 17.3 |
| ILSVRC AlexNet [KSH12] | 227.5 | 56.9 | 7.1 |
| MNIST MLP [CHS16] | 147.2 | 36.8 | 4.6 |
| MNIST MLP Small (This work) | 0.40 | 0.10 | 0.01 |
| CIFAR-10 CNN [CHS16] | 56.1 | 14 | 1.7 |
| CIFAR-10 CNN Small [LSC18] | 0.36 | 0.09 | 0.01 |

method allows further model size reduction and speedup compared to a binarized network. 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 [PGC17].
- 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 very low memory footprint and runtime. 3PXNet imple-

mentation 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 Related Work

### 2.2.1 Binarized neural networks

Reducing the computational complexity of Neural Network training and inference has become a major research topic in recent years [CWZ17]. Multiple works have shown that decreasing the precision of underlying computation through quantization does not affect accuracy, while significantly improving storage and runtime [VSM11, CCR13, GAG15, ZWN16, JAM17, JBB17]. To reduce the computational complexity to its limit, researchers have proposed a variety of implementations that binarized both weights and activations [CSM15, CHS16, KS16, ROR16, CHS16]. Those Networks are commonly referred to as XNOR-Nets because multiplication 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. [UFG17] have created FINN, a framework for binarized FPGA accelerators, which was further expanded to larger models by Fraser et. al. [FUG17]. Other binarized accelerators have been proposed, both targeting FPGAs [LLX17, ZSZ17, LXY17, YHF18], ASIC [ACR18, BKA18, CSB18, SNT18], and in-memory compute [JKW17, BYM18]. Yang et. al. [YFB17] have developed BMXNet, an extension of MXNet [CLL15] based on a binarized GEMM kernel. Depth-first binarized convolution implementations have also been shown for both CPU and GPU [HZL18, PTT18, MTK17]. Our implementation leverages the same depth-first approach to convolutional layers, while also incorporating coarse intra-kernel pruning.

### 2.2.2 Weight pruning

Weight pruning in Neural Networks was first proposed over 20 years ago as a way of improving generalization and reducing computational complexity for both training and inference [CDS90, HSW93]. Recently, Han et. al. [HPT15] 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 [HMD15]. However, deploying pruned models on highly-parallel architectures has proven problematic due to storage overhead and irregular memory access patterns of sparse matrix multiplication [YLP17, SCS17].

To make pruning more regular, multiple forms of "structured" pruning have been proposed. Lebedev and Lempitsky [LL15] proposed group-wise sparsification. Foroosh et. al. [FTP15] hard-coded the sparsity patterns into the source code, achieving up to 6.88x speedup on CPUs. Anwar et. al. [AS16, AHS17] explored different granularities of pruning: feature map, kernel, and intra-kernel and introduce kernel strided sparsity. Sredojevic et. al. [SCS17] have proposed an algorithmic way of inducing regularity in sparse networks. Yu et. al. [YLP17] have developed a hardware-aware pruning method called Scalpel, which matches the coarseness of pruning to the parallelism of the underlying hardware. Our approach to packing is based on Scalpel but applied to binarized models and uses CPU bitwidth as packing granularity, while also permuting layer inputs to improve packing opportunities. Wang et. al. [WZW18] 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 [CKE16, ZDZ16, PRM17, JDS17]. Crossbar-aware pruning has also been proposed given the recent emergence of analog crossbar-based accelerators [LDZ18].

### 2.2.3 Sparse binary networks

While traditional implementations of ternary networks can be considered binary-sparse, 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) [ZHM16, LZL16, ALP17, MKM17, KBM17, YSN18, HGF18]. Thus we would like to make a distinction 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. [LXZ17], exploited Singular Value Decomposition to reduce kernel sizes in BNNs. Certain software and hardware implementations rely on operand-gating XNOR multiplication [HBO18, DJP18], however, they still require 2 bits of information per weight: value and mask. Li and Ren [LR18] 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. [FFG17] have discussed the implications of exploiting sparsity in binarized FPGA accelerators, but not software ones.

### 2.2.4 Machine learning on embedded systems

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 the 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 [IHM16] and MobileNets [HZC17] have been developed, specifically targeting low memory footprints. Lai et. al. [LSC18], 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 [KGV17, GSG17].

## 2.3 The 3PXNet Approach

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

### 2.3.1 XNOR networks

Binarized neural networks reduce network size by having only one bit for each weight, allowing multiple weights to be packed in a binary vector, e.g. a processor 32-bit word. By constraining the 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 popcount 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.

### 2.3.2 Challenges in pruning XNOR networks

Dense binarized XNOR networks are relatively straightforward to pack for both weights and activations thereby getting close to the "32x" leverage [CHS16]. However, introducing sparsity on top of binarization will not necessarily improve the results further. We first illustrate how naively pruning binarized networks worsens storage and runtime. Consider a "small" binarized MLP used for MNIST (see Table 2.1) 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. If we prune it without any constraints and store the sparse weight matrix using compressed-sparse-row (CSR) format, binary weight packing and "SIMD" XNOR multiplication cannot be leveraged easily. 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 will have a more profound impact on high levels of sparsity, 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 implementation. 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.

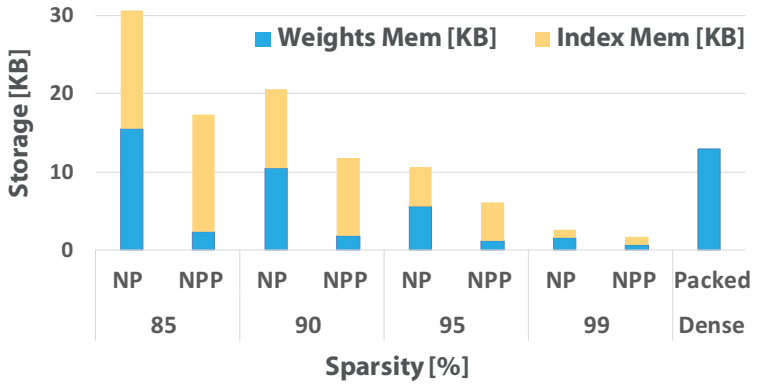



Figure 2.1: Storage requirements for Dense, NP, and NPP XNOR "small" MNIST MLP for varying levels of sparsity.

While NPP 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), the 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 clearly shows that naively pruning

binarized networks, even with packing, is not beneficial at best, and possibly detrimental to both their size and runtime.

### 2.3.3 Pruning a packed XNOR network

To make the 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 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. 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, we propose to permute the weight matrix so that weights inside the 32-bit packs are more likely to be all zeros in the ternary network. Figure 2.2 illustrates the effect of permutation on packing for a pack size of 4. A similar approach is also used in [LDZ18] 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. 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 choice will result in some weights being forced to change. On the other hand, positions filled

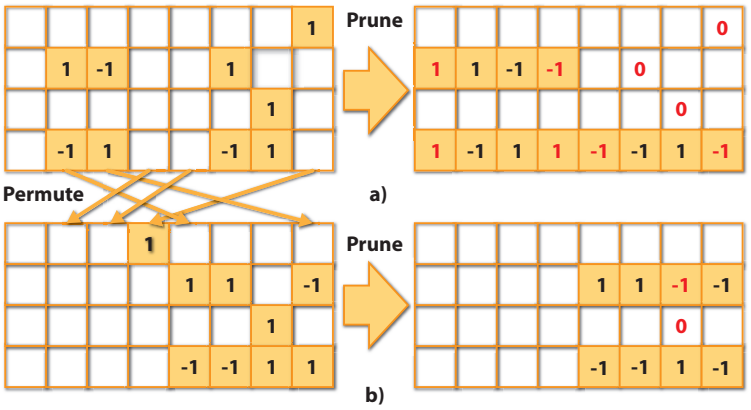



Figure 2.2: Pruning with packing constraint of 4 bits a) without permutation, b) with permutation

with 0s in a pack will 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.

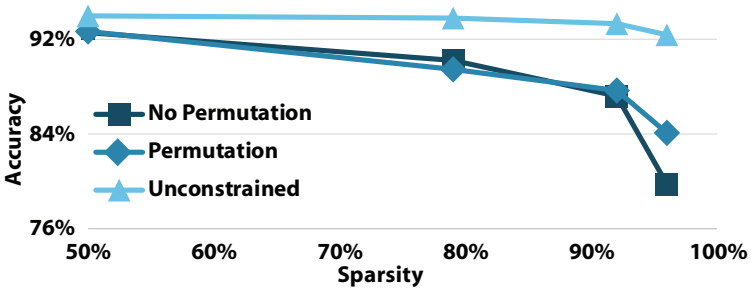



Figure 2.3: Comparison of training results with and without permutation for different sparsities.

### 2.3.4 Training 3PXNet

Network pruning is usually performed by training a dense network and then deleting some edges or kernels. Since 0 cannot be represented in a binarized 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 from the one in [CHS16] where full-precision weights are kept during training and binarized during inference, except that the weights are ternarized using a threshold function instead of binarized.

Listing 2.1: Pseudocode for calculating threshold for NPP.

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

The threshold used 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 constraint 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.

Listing 2.2: Pseudocode for calculating threshold for packed pruning.

```
weight_split = split(weight, split_size=word_width)
// For every split weight pack
for sp = 0 to size/word_width
    weight_sum[sp] = sum(abs(weight_split[sp]))
weight_sorted = sort(weight_sum, descending=True)
index = ceil((1-sparsity) * size / word_width)
thres = weight_sorted[index-1]
```

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 prunes 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 CNN we are not pruning the input layer, so permutation doesn't change the achievable topology. Because 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 the result.

## 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 [STM18], and a Raspberry Pi, with the configurations shown in Table 2.2.

Table 2.2: Hardware platforms used for the runtime experiments. Only the *Small* platform has a DSP extension with hardware multiply-accumulate unit. All three microcontrollers are from the ST Nucleo family.

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

### 2.4.2 Benchmarks

We evaluate our approach using two fully-connected networks (MLP) and a small convolutional neural network (CNN) on the 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 10 classes. MLP-Large (MLP-L) and CNN-Large (CNN-L) are used in [CHS16]. 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 [LBB98], 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 [War18]. Networks are trained with PyTorch 0.4.1. All models are trained on the training set provided, and accuracies are measured on the testing sets. We used Adam optimizer [KB14] for training, with a batch size of 256 and an initial learning rate of 0.001.

Table 2.3: Benchmark models and datasets

| Dataset | Model | | Architecture |
|---|---|---|---|
| MNIST | MLP | Large [CHS16] | 784-4096-4096-4096-10 |
| | | Small | 784-128-10 |
| | CNN | Small | $32CONV5-MP2$ |
| | | | $32CONV5-MP2$ |
| | | | $10FC$ |
| CIFAR-10 | CNN | Large [CHS16] | $128CONV3\times2-MP2$ |
| | | | $256CONV3\times2-MP2$ |
| SVHN | | Medium | $512CONV3\times2-MP2$ |
| Speech[War18] | | | $(1024FC\times2)-10FC$ |

### 2.4.3 Baseline

On Nucleo boards, we use the ARM CMSIS-NN [ARM19], version 5.3.0 optimized for Cortex-M processors. On Raspberry Pi we use Arm Compute Library [ARM18], version 18.03, with NEON extension enabled 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. 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

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) |

Figure 2.4 compares accuracy vs. model size of 3PXNet to a dense binarized network eBNN [MTK17]. 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, additional size compression comes from tweaking the network structure itself, whereas for us it comes from pruning.

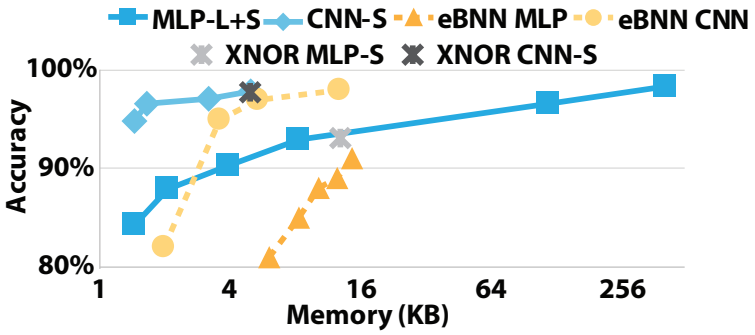



Figure 2.4: Accuracy vs. Memory tradeoff compared to eBNN and dense XNOR.

Training results are shown in Table 2.4. Each network is trained with two sparsities, 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 to 50% sparsity due to their negligible impact on storage and computation. Because we're 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 don't decrease in size with higher sparsity, the actual compression rate varies a lot across models and we think it's 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.

Network pruning can also be performed on higher precision models, so we compared our performance to magnitude-based weight pruning [HPT15]. 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 2.5 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 [YLP17], but are mostly limited to 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.

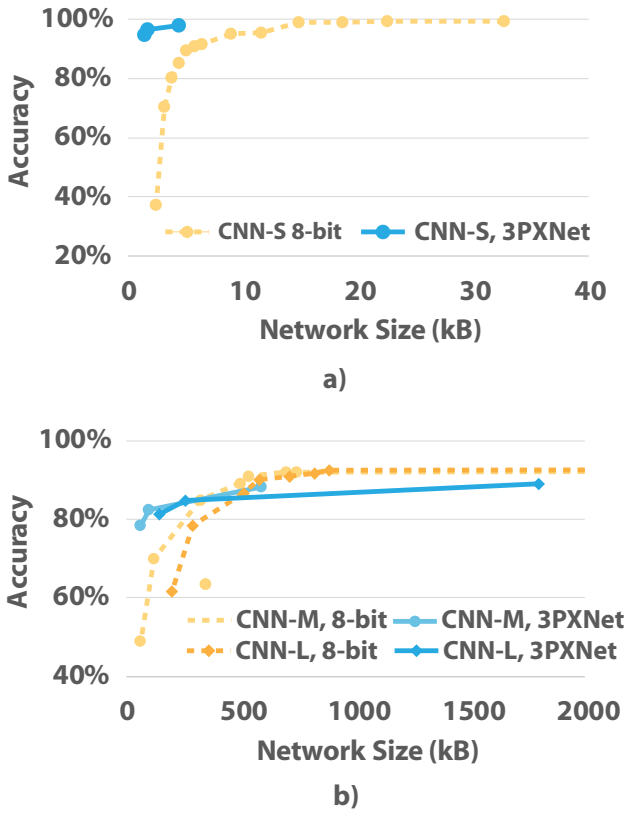



Figure 2.5: Accuracy comparison between sparse 8-bit network and 3PXNet, for MNIST (a) and CIFAR-10 (b).

### 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 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 one the MLP Large only after binarizing and sparsifying the network. On Raspberry Pi, the speedups obtained for both MLP-S and MLP-L are much larger than what we'd expect (76x-400x). For the former, the model is so small 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 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 at all. On Raspberry Pi, 3PXNet achieves a 7.4-10.4x speedup vs ARM CL and a 2-2.75x speedup over dense binary. Energy reduction is proportional to runtime, which means 3PXNets could greatly extend the battery life of edge devices.

A few factors are limiting achievable speedups, for both Dense 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 [MKL18]. 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. We plan on exploring fixed-point batch normalization in the future to alleviate that. CNN speedups are heavily limited by 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 $3\text{PXNet}_{\text{high}}$ that number reaches 92%.

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 Conclusion

In this chapter, 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 complex deep learning models on the smallest of 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 `https://github.com/nanocad-lab/3pxnet`.

# CHAPTER 3

# Making Stochastic Computing Work for Deep Learning[1]

With the exploratory work on binarization and sparse binary networks finished, several issues arise from binarized neural networks. The first is accuracy. Binarized neural networks struggle to reach similar accuracy as 8-bit fixed-point models at the same model size. While methods like expanding the layers and replacing the first and last layers with 8-bit computation can improve accuracy, they also diminish the benefits of binarized neural networks. As such, there needs to be another way of scaling up binarized neural network accuracy without (significantly) expanding the model size. The second is performance. Binarized neural networks are initially designed to reduce the cost of multiplication. This is a reasonable point for optimization, as an 8-bit fixed-point multiplier is roughly 8 times more expensive compared to an 8-bit adder, and the difference is even larger for higher-precision multiplication. However, when both activations and weights are reduced to single bits and multiplications are simplified to a single XNOR gate, additions become the main performance bottleneck. This effect can be seen both theoretically and in practice. Whereas multiplications are simplified to a single gate, additions are still assumed to be accurate (before binarization at the end of the multiplication). Accurate addition of two 1-bit multiplication results requires a half adder, which is more expensive than an XNOR gate. What complicates the problem even

[1]**This work is performed in collaboration with Wojciech Romaszkan, Tristan Melton, Sudhakar Pamarti, and Puneet Gupta. This chapter contains material previously used in [RLM20]. My contributions to this work include determining the design choices for SC-based neural networks and training the models.**

further is that the cost of addition does not scale linearly with the size of the accumulation. Whereas adding two 1-bit numbers requires a half adder, adding another 1-bit number on the result of a 256-way addition requires a lot more, as the resulting carry needs to propagate through an 8-bit intermediate result. The same effect can be seen in the instruction throughput of common processors. For Intel processors, different versions of vpadd (256-bit packed addition of integer values) execute at the same rate as vpxor (256-bit packed xor of integer values). As the number of elements calculated by a single instruction can be calculated as vector width/width of a single element, additions of 16-bit numbers execute at half the rate (16 additions per cycle) of 8-bit numbers (32 additions per cycle), and 1/16 the rate of xnor multiplications (256 multiplications per cycle). As a result, the performance of binarized neural networks becomes bottlenecked by **additions** instead of multiplications. This is likely a major reason for the subpar performance of existing software binarized neural network implementations compared to their 8-bit counterparts (other than the difficulty of reaching high utilization due to the reduced problem size). Pruning the model can alleviate this issue by reducing the size of the accumulation, but the saving is limited unless the sparsity is very high. For instance, a layer with a 4096-way dot product would require a 16-bit accumulator to avoid overflows. Simplifying it to 8-bit accumulations would require reducing the dot product size to $\leq 256$, which equates to 94% sparsity.

With the issues of binarized neural networks in mind, stochastic computing (SC) becomes an attractive choice to extend the accuracy of binarized neural networks while retaining its benefits. Stochastic computing promises high area efficiency and high system efficiency with reduced memory accesses stemming from the area efficiency. Compared to binarized neural networks, it maintains the single-gate multiplication operators, while also offering single-gate additions. It also allows multi-bit precision on the inputs and weights, which should enable higher accuracy. This chapter provides a brief tutorial on stochastic computing, followed by the basic setup of stochastic computing that forms the basis of SC-based neural networks. This basic setup debuts in ACOUSTIC [RLM20]. It is referred to as OR-SC and will be

further improved in the subsequent chapters.

## 3.1 Stochastic Computing Primer

Stochastic computing represents numbers using randomized bit streams [Gai67]. In an ideal SC bit stream, each of the bits is independent and identically distributed (IID) with a Bernoulli distribution. For the unipolar representation where the representation range is set between 0 and 1, The probability $p$ of the bit being one corresponds to the stream value. For the bipolar representation where the representation range is between -1 and 1, the value of the stream is set to the probability value normalized to the wider range using $v = 2p - 1$. As we will see later, this number representation simplifies some numeric operations between values.

### 3.1.1 Error behavior

For the ideal SC stream described previously, the streams have a well-defined error profile. For a stream with length $n$ and each bit having a probability of $p$, the final stream value is the average of the bits, which is equivalent to the sum of the bits divided by the stream length $n$. The sum of Bernoulli random variables is a binomial random variable, with an average of $np$ and a variance of $np(1-p)$. After dividing by the stream length, the stream value has an average of $p$ and a variance of $p(1-p)/n$. The average value is what we expect, and the variance is a random error term that is value-dependent. The error term has the following properties:

- Large error for values in the middle of the range. The variance function is a concave function of $p$ with maximal achieved when $p = 0.5$. This corresponds to 0.5 for unipolar streams and 0 for bipolar streams.
- Error decreases with stream length. The variance function is a decreasing function

of $n$. If we look at standard deviation which represents the random error, the error decreases at a rate of $1/\sqrt{n}$. On the one hand, this means that SC allows *adjustable precision*. Higher computation precision can be achieved by running the computation for more cycles. The choice of precision can be done during runtime and requires little to no hardware change. On the other hand, this creates a completely different error scaling behavior compared to fixed-point computation. In fixed-point computation, increasing the number representation by 1 bit reduces quantization error by 2. To achieve the same 2X reduction in error, SC requires increasing stream length by 4X.

The two properties listed above provide important guidance to the choice of SC computation elements. The first property means that the number representation should avoid the high error areas for unipolar and bipolar representations. The second property means that SC error scales much slower than fixed-point computation when increasing the number of bits used for representation. This means that SC is not suitable for applications that require high precision (>= 8-bit fixed-point accuracy). The other implication is that optimizations to SC should try to improve effective accuracy for a given stream length (in other words, reduce stream length for the same accuracy) and cannot rely on the reduced error from increasing stream length. Even though the computation performance only contributes to a small portion of the overall system performance, decreasing it by too much will make it the limiting factor.

### 3.1.2 Stream generation

As SC uses an unconventional number representation, it requires conversion to and from conventional number representations like fixed-point or floating-point representations. While some recent works use a sigma-delta modulator [GGS18] to directly convert analog inputs to SC bit streams, most works on SC still require some conversions. We will focus on conversion between SC and fixed-point representation, specifically fractional fixed-point numbers in [0,1)

or [-1,1), as those match the SC representation range. Conversion from SC to fixed-point numbers involves counting the number of 1's in the bit stream, which is typically implemented as a counter. Conversion from fixed-point to SC is more complicated and is usually referred to as stream generation.

Stream generation requires two components. A typical stream generator requires a random number generator (RNG) and a comparator. An RNG generates random variables between 0 and 1 for unipolar streams and -1 and 1 for bipolar streams. The random number is compared with the fixed-point number. If the fixed-point number is larger than the random number, the stream value is 1 for that cycle. Otherwise, it is 0 for that cycle. The comparator can be replaced with a multiplexer (MUX) tree. MUX tree and comparator generate different streams from the same RNG, but the error and performance properties are similar between the two, so we use comparators due to their ease of modeling in software. An ideal SC stream requires independent and identically distributed random variables as the RNG, which equates to a true random number generator (TRNG) to avoid correlation between streams. TRNG is expensive to implement in hardware, so most works on SC opt for cheaper pseudo-random number generators (PRNG).

### 3.1.3 Computation

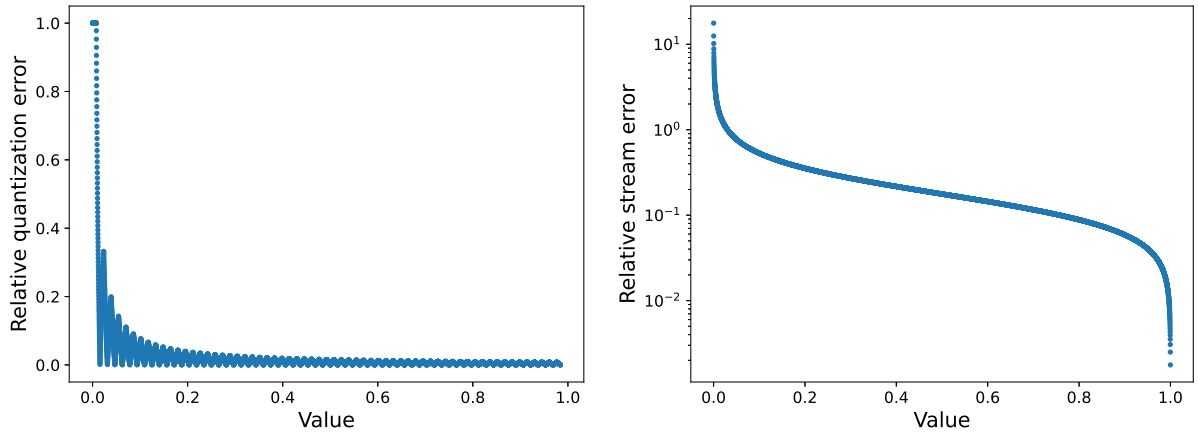
The major benefit of SC is the compact computation units, especially for multiplications and additions. Multiplications are performed using bit-wise AND (unipolar) or XNOR (bipolar) gates. If streams are processed in time (each cycle processes one bit of each input), multiplications can be performed using single gates. This compact representation is the most important benefit of SC and needs to be fully exploited.

Additions are more complicated for SC, and can be broadly divided into two categories:

1. SC additions perform bit-wise additions in SC to maintain the compactness of addition and SC stream output.

2. Fixed-point additions perform additions using fixed-point adders to maintain accuracy.

Due to the limited representation range of SC, SC additions run into saturation issues. The most common SC addition uses multiplexers (MUX). For an n-input MUX using a select signal that randomly and uniformly selects between all n inputs, a MUX avoids saturation by generating the average of the n values. This scaling behavior is okay for adding a few numbers. However, the scaling effect becomes prominent for large n and is especially problematic when the number of inputs exceeds the stream length.




(a) (b)

Figure 3.1: Relative error resulting from (a) quantization and (b) randomness of stochastic computing.

To illustrate this scaling effect, we consider the use of MUX-based adders in dot products. SC dot products using MUX-based adders typically come in two variants. The first is where multiplications and additions are separated. Multiplications are performed using a traditional SC multiplier in the form of an AND or XNOR gate, and additions are performed using MUX with a random signal that uniformly selects between the different products. We denote this formulation of MUX-based adders as the multiply-add version of MUX, or M-MUX in short. For M-MUX, the output of a dot product can be formulated as M-MUX$(a, b) = 1/n \sum a_i b_i$, where $a$, $b$ are the input vectors, and $n$ is the size of the

dot product. The M-MUX adder thus scales the multiplication result by $1/n$ before adding them together. To understand how MUX downscaling affects the error of accumulation, we consider two components that contribute to the output error. The first is the quantization error. Stochastic computing typically utilizes uniform quantization to take advantage of the probabilistic compute units. Uniform quantization has consistent absolute quantization error regardless of magnitude. As such, the relative error from uniform quantization is larger for smaller values, which can be observed in Fig. 3.1a. The second component is the random error from stochastic computing. If streams are generated using a true random number generator (TRNG),[2] the expected output error can be derived from a binomial distribution and is equal to $E_{abs} = \sqrt{\frac{v(1-v)}{n}}$, where $v$ is the expected output value and $n$ is the stream length. The expected error is a concave function of $v$ with a small error for values close to 0 and 1, but the relative error $E_{rel} = \sqrt{\frac{1-v}{vn}}$ is a decreasing function of $v$ and is large close to zero, as is shown in Fig. 3.1b. Both components increase relative error for values close to 0, which M-MUX accumulation enforces for large $n$ values. Since M-MUX accumulation results need to be scaled up by $n$ to recover the true accumulation results, the large relative error translates directly to a large absolute error. For neural networks with dot products going up to the thousands, MUX accumulation leads to a significant loss in accuracy, as the scaling factor is the same size as the dot product size, Fig. 3.2 compares the accuracy of MUX- and OR-based accumulation (to be introduced later) for 1000-wide accumulation. Inputs to the accumulation are sampled such that the sum of all inputs has a variance of 0.5 and a mean of 1. Since neural networks are typically initialized to maintain unit variance throughout the model, it is a valid assumption to have 0.5 variances for half (positive values) of the inputs. Due to the scaling effect, M-MUX requires very long streams to achieve reasonable accuracy.

The other form of MUX addition fuses the multiplication component into the MUX adder and utilizes the select signal to achieve a weighted sum between inputs, which we denote as

[2]TRNG is used due to the simplicity of modeling, but other RNG sources and SC computation results show similar trends

the weighted-add version of MUX, or W-MUX in short. The output of a dot product for this implementation can be formulated as

$$\text{W-MUX}(a, b) = \frac{\sum a_i b_i}{\sum |b_i|} \tag{3.1}$$

where $b$ is the weight vector. The select signal for the weighted sum can be further simplified to a counter as proposed in the CeMux implementation in [BH22]. Compared to the multiply-add version which scales all multiplication results by $1/n$, the weighted-add version alleviates scaling by normalizing the weight values to $\frac{b_i}{\sum |b_i|}$, as can be seen in Eq. 3.1. However, this also means that the weighted-add version does not fully resolve the scaling issue, as the weight values have an average magnitude of $1/n$ after normalization. With weight and output quantization considered and the output scaled back to the original range, Eq. 3.1 becomes

$$\text{W-MUX}(a, b) = (\sum |b_i|)\text{Quant}(\sum a_i \text{Quant}(\frac{b_i}{\sum |b_i|})) \tag{3.2}$$

"Quant" is the quantization function that rounds or truncates the weights and addition results to $log2(b)$ bits, where $b$ is the bitstream length. This modeling represents a best-case scenario for W-MUX and ignores any additional error from SC computation (such as the ones shown in Fig. 3.1b). MUX_Ideal in Fig. 3.2 depicts this quantization effect assuming all weights are one. All weights being one is only one case and weight values are typically different in actual applications, so we considered its performance when training a neural network. We trained a small 4-layer CNN [LSC18] on the CIFAR-10 dataset using the idealized model of the W-MUX adder as shown in Eq. 3.2. Due to the $1/n$ average weight value, *the stream length needs to be at least the size of the dot product for reliable representation of the weight values.* Since the largest dot product in the model is 1024, W-MUX fails to converge reliably and always drops to 10% accuracy (equivalent to random guess) either from the beginning (for stream length $\leq 128$) or after a few epochs (for stream length $\leq 512$). Since prior works on MUX-based addition can be categorized either into M-MUX or W-MUX and both suffer from the scaling issue, it is difficult to achieve good accuracy with reasonable stream length using MUX-based additions.

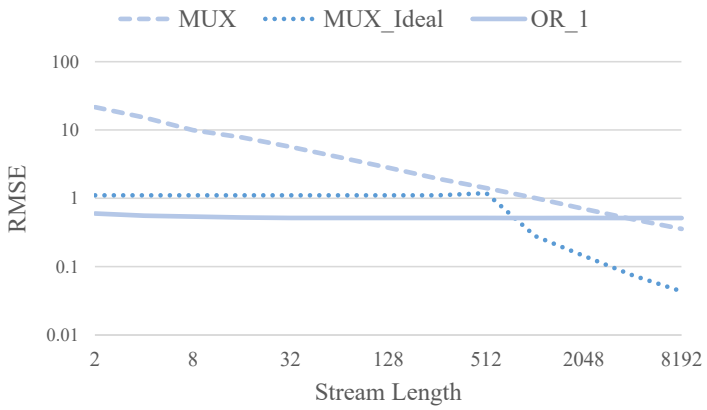



Figure 3.2: Comparison of MUX- and OR-based adders' error with respect to accurate addition.

An alternative way of stochastic accumulation, OR-based accumulation has been proposed in [DMC93]. It performs additions between inputs using a bit-wise OR gate. The OR gate performs accurate addition if 0 or 1 of inputs is 1, but saturates to 1 if more than 1 input is one. It is scaling-free (important for very wide 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 [RLD17]. The other issue with it is that it is not an exact addition. For a two-input OR, the result is equal to $v_1 + v_2 - v_1 v_2$ instead of $v_1 + v_2$.

Compared to MUX-based and OR-based adders, fixed-point adders achieve accurate or near-accurate summation between inputs, making them the most common choice of addition implementation in recent SC works [SL17, HGT19, WLY20]. However, they are expensive to implement since full adders required in approximate/accumulative parallel counters are larger, slower, and less energy efficient than multiplexers or OR gates[LLR18].

### 3.1.4 Storage

Stochastic computing has inefficient storage by nature. Representing a k-bit fixed-point number requires a stream length of $2^k$. This means it is generally not desirable to store the SC streams. Instead, the streams should be converted to fixed-point numbers for applications that require storing intermediate results.

### 3.1.5 Non-linear computation

The previous computation section focuses on multiplications and additions using stochastic computing and ignores non-linear computations which are also common in neural networks. While previous works have shown that it is possible to perform non-linear operations like divisions, hyperbolic tangent (Tanh), and rectified linear (ReLU) using finite-state machines (FSM), performing these operations in SC does not offer significant performance improvements over their fixed/floating-point counterparts. As conversions to fixed-point are needed for storage efficiency reasons, we perform these non-linear operations using fixed-point operators and avoid performing them in SC.

## 3.2 Stochastic Computing Baseline Implementation

With the SC introduction in Sec. 3.1, the following section introduces the initial SC implementation aimed at neural network inference acceleration in [RLM20], which tries to avoid some of the previous-mentioned pitfalls. Most of the components introduced here will be refined in future chapters, but the design choices here act as a baseline.

### 3.2.1 Split-unipolar stream representation

In neural networks, maintaining high accuracy mandates using weights with both positive and negative values, which makes bipolar representation the most common choice when im-

plementing SC-based accelerators [RLD17, SNL17, LLR18]. However, [YKL17] noticed that using unipolar representation results in higher accuracy compared to bipolar. This is because 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) value of error can be represented as:

$$E_u = \frac{\sqrt{n_u v(1-v)}}{n_u} = \sqrt{\frac{v(1-v)}{n_u}} \tag{3.3}$$

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

$$E_b = \sqrt{\frac{1-v^2}{n_b}} \tag{3.4}$$

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 $0 < v < 1$. This means that bipolar representation will always require $> 2$ times the bit stream length to represent the same value with the same accuracy compared to unipolar representation.

To capitalize on both shorter streams offered by the unipolar representation, as well as the ability to represent negative weights by the bipolar representation, 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 in the previous layer, they can be represented using 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. While enforcing half of the representation to zeros seems wasteful and indicates that half of the hardware for weights is always wasted, this representation ensures that all weight values are represented in the same

format and makes it easy to implement in hardware.

### 3.2.2 Computation skipping for average pooling

While multiplexer-based average pooling has been used in prior SC work [LRL17, LLR18], to the best of our knowledge we are the first ones to exploit its unique properties to perform computation skipping. The average pooling multiplexer selects inputs based on a random select signal, which is the 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 issue with the computation skipping scheme is that the results are correlated, thus necessitating the randomization of output streams. This issue is avoided by converting the stream to fixed-point values after each layer.

### 3.2.3 Making OR accumulation practical

As mentioned in Section 3.1.3, OR accumulation has been largely disregarded by prior work, due to its inherent inaccuracy for bipolar streams. Given our use of split unipolar streams, we revisit OR. Figure 3.3 a) shows the accuracy comparison between MUX and OR for accumulating $3 \times 3 \times 256 = 2304$ random numbers to 1. Since neural networks generally require large matrix multiplications which have very wide additions, MUX is not suitable for compact SC addition. As mentioned earlier, using fixed point binary adders is undesirable due to the large area overhead.

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. This, in turn, requires multiplications in the neural network to be performed explicitly while training and also slows down addition during both forward and backward passes (∼15X longer training runtime). For OR-based addition with inputs $a_k$, the expected output value is equal to

$$OR(a_1, a_2, ..., a_n) = 1 - prob(0\ input\ is\ 1) = 1 - \prod(1 - a_k) \tag{3.5}$$

By assuming all inputs are the same, Eq. 3.5 simplifies to $1 - (1 - s/n)^n$, where s is the normal sum of the inputs. When n is large, it further simplifies to

$$OR(a_1, a_2, ..., a_n) \approx 1 - e^{-s} \tag{3.6}$$

This approximates OR addition by adding an activation function after the normal network layer. To show the validity of this approach, we extracted run-time results from training with OR-addition, and results are shown in Figure 3.3 b). OR-additions have very small variation 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.

### 3.2.4 Modeling the error of stochastic computing

The error introduced by SC greatly affects the accuracy of a network if not taken into consideration. A network trained without error assumption can lose $> 70\%$ accuracy when validated using SC. This necessitates training with error injection or training with stochastic bitstreams directly. To model SC computation error, we introduce an error term that has a Gaussian distribution with a standard deviation equal to the error defined in Equation

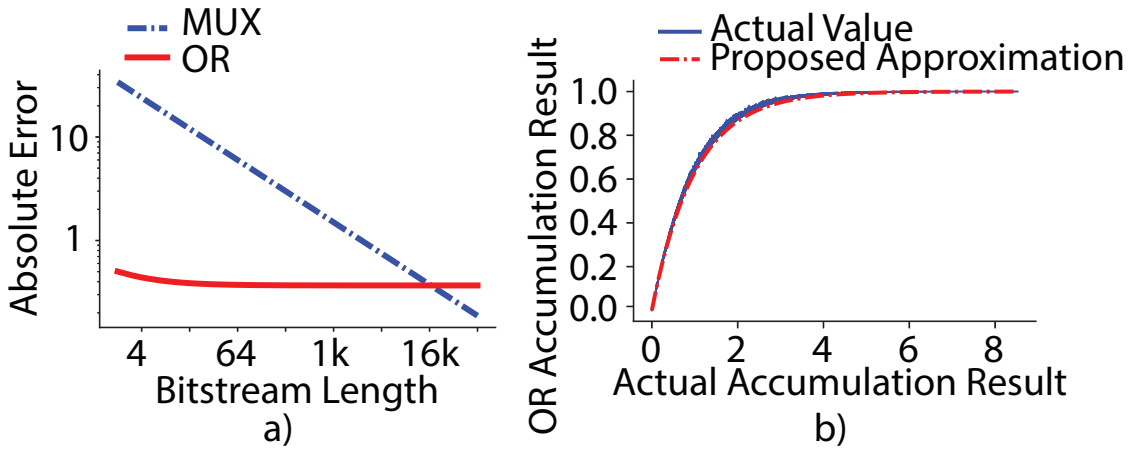



Figure 3.3: Accuracy comparison between MUX and OR a) and comparison of approximation methods for OR accumulation b).

3.3[3]. This error term is added to every positive and negative MAC result before they are converted to binary for subtraction. The effect of this error injection is shown in Table 3.1. The bit stream length used is 256, and the network used is a small CNN used in [LSC18]. Training with error injection significantly increases SC validation accuracy.

Error injection doesn't fully simulate the effect of SC due to the use of approximate OR. To eliminate this gap we attempted to train with SC bitstreams directly. The forward pass of training is performed using simulated SC, while the backward pass is performed using the approximated OR accumulation. The results of this method are shown in "SC" rows in Table 3.1. Using SC directly during training is the best simulation of the actual scenario, so it has the best accuracy. However, there is a large drop in training speed, and training with SC directly is at least 10X slower than training with approximated OR accumulation, which renders it useless for larger datasets like ImageNet.

[3]Since different bits in a bit stream are identically and independently distributed, the central limit theorem holds and Gaussian distribution is a valid approximation for relatively long bit streams.

Table 3.1: Validation accuracy of a 4-layer CNN[LSC18] for SC hardware with different training methods.

| | CIFAR-10 | | | SVHN | | |
|---|---|---|---|---|---|---|
| Training | Normal | Error | SC | Normal | Error | SC |
| Accuracy[%] | 24.03 | 67.44 | 74.9 | 34.33 | 84.96 | 86.75 |

## 3.3 Conclusion

This chapter gave a brief introduction to the relevant aspects of stochastic computing and described an implementation of SC-based neural networks used in ACOUSTIC [RLM20]. From the properties of stochastic computing, we understand that SC-based neural network acceleration needs to exploit the high density offered by SC while minimizing its issues like accuracy deficits, long representation, and inefficient non-linear operators. The OR-SC implementation uses the split-unipolar representation and skip-based average pooling to reduce stream length and introduces training modifications to make OR-based accumulation practical to maintain SC's high density.

# CHAPTER 4

# Improving SC Generation and Execution[1]

While the implementation in Chap. 3 is a reasonable starting point, the accuracy of neural networks using OR-SC remains low, and long streams are required to achieve acceptable accuracy. Although computation cost is relatively low compared to memory cost in neural network accelerators, this applies mainly to fixed-point accelerators. As SC reduces memory cost with its increased reuse, computation cost also becomes significant, and long stream lengths affect the overall efficiency of SC accelerators. The inaccuracy of OR-SC originates from stream generation and computation. While not detailed in Chap. 3, stream generation in OR-SC uses 16-bit linear feedback shift registers (LFSR). Using a relatively long LFSR reduces the chance of correlation between streams, but also increases stream generation cost and makes it more likely to have errors in short streams. Computation in OR-SC uses OR gates as the accumulator. OR gates are cheap to implement but have limited output range and precision. This chapter will introduce 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. Our contributions are as follows:

- We show that, with appropriate training, neural networks can learn the biases caused by the use of pseudo-RNGs. Extensive sharing of pseudo-RNGs in SNGs can *improve*

[1]**This work is performed in collaboration with Wojciech Romaszkan, Sudhakar Pamarti, and Puneet Gupta. This chapter contains material previously used in [LRP21]. My contributions to this work include the concepts of co-optimized shared generation and training and partial binary accumulation, alongside accuracy results from training.**

accuracy compared to using non-shared TRNGs by as much as 6.1% points while reducing energy and area.

- 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 still maintaining 2.2-4.0% points accuracy advantage over prior state-of-the-art SC implementations.

## 4.1 Co-optimized Shared Generation and Training

RNG Sharing was believed to be detrimental to stochastic computing accuracy[IIS14, NPH17], 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 the same output streams from the same fixed-point inputs, enabling the model to train for a fixed, instead of random error. We achieve determinism using maximal-length linear feedback shift registers (LFSR) as RNG. When generating streams of length $2^n$, an n-bit maximal-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[LSC18] on the Street View House Numbers (SVHN) [NWC11] dataset. Streams are represented using the split-unipolar format, and OR gates are used for accumulation, similar to [RLM20]. 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. 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.

The accuracy drop from extreme sharing also means that low discrepancy (LD) sequences [LH17] are not suitable for OR accumulation due to the difficulty of generating multiple uncorrelated streams (e.g., 5x5x32 kernels are equivalent to 800-dimensional random numbers, at which point LD sequences offer little benefit). 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%. Subsequent results in GEO will use the moderate sharing scheme (up to the limit of availability of unique RNG seeds).

## 4.2 Partial Binary Accumulation

Many recent SC works opt to perform accumulations in the fixed-point domain, as it offers higher accuracy than SC-based addition. In contrast, a 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

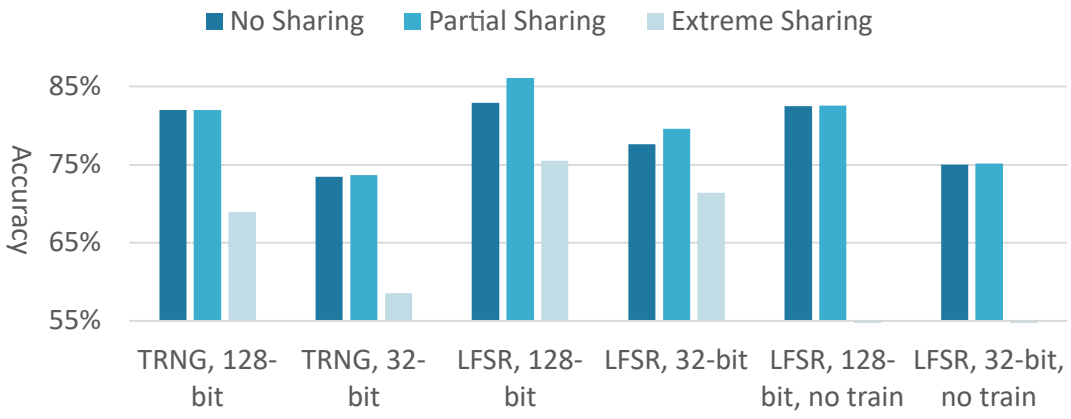



Figure 4.1: Accuracy vs. sharing for TRNG and LFSR-based random number generation.

and computing the remainder of the accumulation.

The partition between SC and fixed-point accumulation significantly affects both accuracy and performance. Using an approximate parallel counter (APC) [KLC15], for instance, allows one layer of SC accumulation before fixed-point accumulation. The SC addition in APCs uses a combination of AND and OR gates. This makes it equivalent to multiplexers in scaling behavior and is thus unsuitable for multiple layers of accumulation. Using OR for accumulation with training allows an arbitrary trade-off between SC and fixed-point accumulation. We tested model accuracy with different fixed-point accumulation levels using the same setup as in Sec. 4.1. 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) as well improves accuracy by $< 0.5\%$ but increases the number of fixed-point adders by 5X for $5 \times 5$ 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 4.3, we show that those changes have minimal impact on the overall architecture.

Figure 4.2 shows the overhead, in terms of area, of implementing SC-based MAC units with 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 the 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 4.2 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 [KLC15] (APC) offer 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.

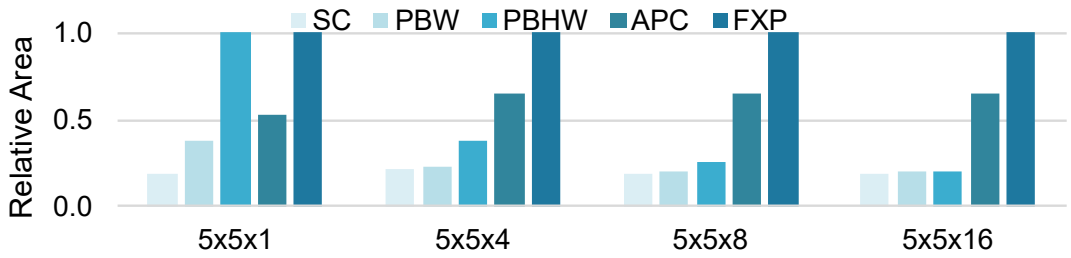



Figure 4.2: Area comparison for different hardware implementations of SC-based MAC units for different kernel sizes and different levels of partial binary accumulation.

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 Results

### 4.3.1 Evaluation methodology

We evaluated the accuracy of GEO on CIFAR-10, SVHN, and MNIST datasets. For CIFAR-10 and SVHN, we use the same 4-layer CNN [LSC18] (CNN-4) as in Section 4.2 and VGG-16 [SZ14]. 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 [LBB98]. We use PyTorch 1.5.0 to train the models. Models are trained using an ADAM optimizer 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. Due to the use of computation skipping for average pooling, layers with pooling can use shorter streams. Output layers always use 128-bit streams due to their small performance impact but noticeable accuracy benefits from longer streams. The stream length used in hardware is double the specified value due to the use of split-unipolar representation.

As a fixed-point baseline, we use Eyeriss [CKE16], 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 [GH17]. For SC comparison points, we use the ACOUSTIC [RLM20], Sign-Magnitude SC (SM-SC) [ZLS18] and SCOPE [Li18]. ACOUSTIC configurations are sized to have the same amount of memory and computing as GEO. 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 [Li18]. 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 [SB17]. We further compare GEO-ULP with CONV-RAM [BC18] and MDL-CNN [SFN19] mixed-signal accelerators. To ease future comparisons and benchmarking, we open-source our SC training code (heavily optimized for stream-based training on GPUs and CPUs) at `https://github.com/nanocad-lab/geo`.

### 4.3.2 GEO accuracy comparisons

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

| | | Eyeriss | | ACOUSTIC [RLM20] | | GEO | | | SCOPE[Li18] | CONV-RAM[BC18] | MDL-CNN[SFN19] | SM-SC[ZLS18] |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 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% | — |

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 [RLM20] and similar accuracy at the same stream length compared to [ZLS18].* 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

[2]The original Eyeriss [CKE16] uses 16-bit inputs 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.

1.9% lower when using 64-128[3]. Accuracy with VGG-16 is 2.2% lower than the 8-bit fixed-point baseline on CIFAR-10 and comparable on SVHN. Accuracy on MNIST is already comparable to fixed-point in the baseline, and GEO optimizations don't affect it. Compared to CONV-RAM[BC18], an in-memory architecture and MDL-CNN[SFN19], a time-domain architecture, GEO offers superior accuracy even with 16-32 stream length.

### 4.3.3 Performance Impact of GEO Enhancements

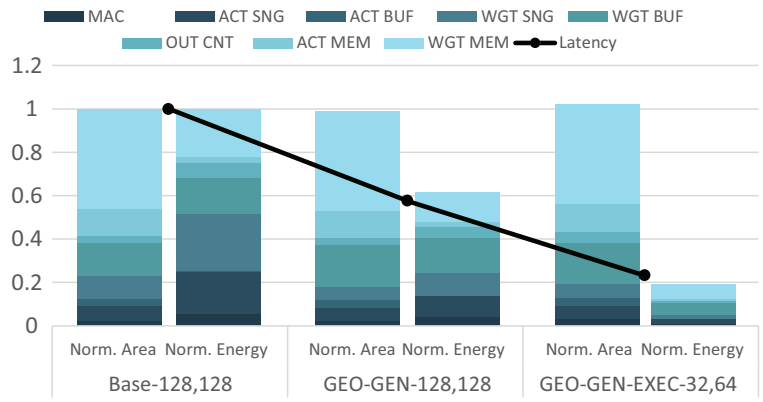



Figure 4.3: Area, energy, and latency comparison between baseline and different GEO configurations (normalized to Base-128,128). The first bar in every group is for area while the second is for energy.

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

- GEO-GEN-128,128 - uses the generation optimizations from Section 4.1.
- GEO-GEN-EXEC-32,64 - uses both the generation and execution (from Section 4.2 optimizations..

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

Both variants include additional optimizations to the SC architecture including progressive shadow buffer, pipelining, and near-memory computation, and more details can be found in [LRP21]. The area, energy, and latency impacts of GEO optimizations on the ULP architecture are shown in Fig. 4.3. 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 area due to progressive shadow buffers is balanced by more extensive RNG sharing. 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, and 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.*

## 4.4 Conclusion

In this chapter, we present GEO, a generation and computation-optimized stochastic computing architecture for neural network acceleration. On the algorithm front, we develop an ensemble of accuracy improvements including co-optimized stream generation and training and partial binary accumulation. Coupled with performance-improving techniques, these optimizations improve accuracy by 2.2-4.0% points compared to previous state-of-the-art SC-based accelerators while also being 4.4X faster and 5.6X more efficient. GEO can compete with fixed-point implementations with similar accuracy and area while delivering up to 5.6X throughput, and 2.6X efficiency gains. GEO, despite being an all-digital, programmable accelerator can achieve efficiency comparable to in-memory/mixed-signal accelerators.

# CHAPTER 5

# Optimizing SC Accumulation[1]

The generation and execution optimizations in Chap. 4 allows a better accuracy-performance tradeoff for SC-based neural networks, but the execution part, in particular, leaves room for improvement. The partial-binary accumulation setup combines OR-based and counter-based additions. This means that the accuracy benefit can be inconsistent based on input distribution. OR-based adders saturate at 1, so any addition results larger than 1 will be capped at 1. The binary addition following the OR gates is supposed to increase the range and alleviate the saturation issue, but its benefits will be reduced if saturation happens within the OR addition tree. Fig. 5.1 demonstrates this behavior. The ideal case in Fig. 5.1a spreads the 1 inputs between the OR trees, and the binary adder avoids saturation and loss of information. However, if the 1 inputs are within the same OR tree as in Fig. 5.1b, the binary addition doesn't help, and the output is the same as a normal OR-based addition. This inconsistent behavior means that the number of binary additions needed may be larger than needed to achieve a specified accuracy level.

The inconsistency in Fig. 5.1 also complicates the training process. The OR-based accumulation introduced in Chap. 3 allows modeling of OR-based accumulation as normal accumulation plus an activation function. However, this method does not apply to partial binary accumulation, as the accumulation is not associative. The accumulation result de-

[1]**This work is performed in collaboration with Wojciech Romaszkan, Sudhakar Pamarti, and Puneet Gupta. This chapter contains material previously used in [LRP23]. My contributions to this work include developing the OR-n methodology and the accompanying training methodology and training the models.**

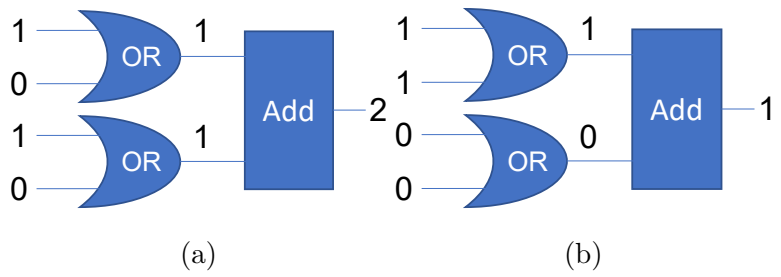



Figure 5.1: (a) best and (b) worst case for partial binary accumulation.

pends on the location of 1's. If the 1's are distributed in different OR trees, the accumulation result is equal to the accurate sum of input bits. If the 1's are concentrated in the same OR tree, the accumulation result is equal to the OR-based addition. For the 5-way partial binary accumulation setup used in Chap. 4, there is a 5X difference between the two cases, and it is impossible to predict the approximate output value given just the sum of inputs. As a result, accumulations need to be broken up in the backward pass. As each partial result of OR-based accumulation needs to be stored for backpropagation, this setup increases memory requirements during training. The reduced dot product size from splitting accumulation also makes it more difficult to achieve good GPU utilization, which affects computation efficiency.

In this chapter, we improve SC accumulation accuracy through Range-Extended Stochastic Computing accumulation (REX-SC), while preserving most of the performance benefits of stochastic computing. We build upon energy- and area-efficient OR accumulation and improve its accuracy by increasing the number of output bits relative to the input bits. Our contributions are as follows:

- We introduce extended range SC (REX-SC) addition methods with higher output precision while preserving the streaming nature of SC.
- We explore the extensive design space of REX-SC accumulation, and develop methods of finding optimal transfer functions.
- We show that REX-SC improves accuracy by 3-8% compared to previous methods

utilizing SC accumulation [RLM20] and reduces energy consumption by up to 3.6X compared to SC with binary accumulation.

- We propose training optimizations to improve the training speed of SC. These include accelerated stream computation simulation, activation calibration, and error injection (a+e).
- We show that our optimizations improve the training speed of SC models using REX-SC accumulation by >22 times compared to previous methods that achieve similar accuracy levels [LRP21].

## 5.1 Motivation

In this section, we motivate the need for new methods of SC accumulation. By analyzing the shortcomings of existing techniques through the lenses of *precision* and *efficiency* we expose a glaring design space gap, waiting to be explored. Most of the previous works on SC focused on stream generation and multiplication [LRP21, HGT19, SL17, WLY20]. Our work aims to mend this oversight, first by showing the importance of optimizing SC addition and then by developing such optimization techniques.

Table 5.1: Summary of various SC addition methods and works using them.

| Type | Streaming | | | Fixed-Point | |
|---|---|---|---|---|---|
| Realization | MUX | OR | **REX** | Parallel Counter | Accumu-lator |
| Precision | Scaled | Approx. | **Approx.** | Exact | Exact |
| Area | Compact | Compact | **Compact** | Large | Large |
| Bipolar | Yes | No | **No** | No | Yes |
| Output Range | Same as Input | | **Higher than Input** | Configurable | |
| Example Usage | [LLR18, RLL17] | [RLM20, LRP21] | **This Work** | [LLR18, RLL17, WLY20] | [SL17, HGT19] |

### 5.1.1 SC accumulation primer

Stochastic computing addition techniques can be divided into two categories: streaming and fixed-point. The former preserves the number representation of the inputs, through the use of single-gate, streaming adder implementations, either a multiplexer (MUX) or an OR gate [AH13]. A MUX-based SC adder uses a random select signal with equal probability for each input, downsampling them to implement scaled addition [AH13]. In comparison, an OR-based adder does not introduce a scaling factor, but it realizes an approximate addition function: $OR(A, B) = A + B - AB$ [RLM20]. The fixed-point addition techniques implicitly convert the streams into the fixed-point domain using parallel counters, accumulators, or a combination of both. Approximate or exact parallel counters add individual bits of multiple stochastic streams. Tab. 5.1 shows an overview of different SC accumulation methods.

### 5.1.2 Precision

SC is an approximate computing method that can introduce random errors during both conversion and computation. Therefore, one of the chief concerns when designing SC systems is their accuracy. While extensive efforts have been directed at reducing the error of both stream generation [LRP21, SL17], and multiplication [SL17, WLY20], they have done little to explore and improve the accuracy of addition operators. The use of multiplexers (MUX) has been largely abandoned for addition because their scaling factors equal the sizes of accumulation [LLR18]. Due to the scaling effect, *the stream length needs to be at least the size of the dot product for reliable representation of the weight values.* Since the largest dot product in the model is 1024, W-MUX fails to converge reliably and always drops to 10% accuracy either from the beginning (stream length $\leq$ 128) or after a few epochs (stream length $\leq$ 512). [2]Deeper neural networks have even larger dot products. The ResNet models

[2]Longer stream lengths may still face convergence issues with W-MUX but are not supported in the SC simulation framework used.

[HZR16] used in Sec. 5.3, for instance, have layers with 3x3x512-size filters, which translates to 4608-sized dot products. Large dot products in turn require longer streams to represent the weight values. As we will show in Sec. 5.3, stochastic computing has trouble competing against fixed-point computation with stream lengths >> 64. Since prior works on MUX addition can be categorized either into M-MUX or W-MUX and both suffer from the scaling issue, it is difficult to achieve good accuracy with reasonable stream length using MUX-based addition.

OR-based adders, while not troubled by scaling factors, come with their own set of issues. First, as mentioned before, they do not implement exact addition. For two inputs $a$ and $b$, an OR gate performs $a + b - ab$ compared to $a + b$ in an exact addition, which leads to output saturation for high-magnitude inputs. While the saturation of OR-based adders can be alleviated with small input values, small input values increase relative error, as is discussed in the previous discussions on MUX. Second, the outputs of OR accumulation have the same precision as the inputs due to the bit-wise computation, which reduces accuracy compared to accumulation without truncation. Recall that adding two N-bit fixed-point numbers requires N+1 bits to avoid overflow. Prior works have shown that algorithms can be trained for approximate addition and saturation for the first and second issues, for example, by using custom neural network training [RLM20, LRP21]. Unfortunately, they cannot correct the loss in intermediate output precision. Other previous works have tried to combine the OR accumulation and fixed-point accumulation to achieve a better trade-off between the two [LRP21]. While it does improve accuracy, combining the two dramatically slows down the training process, as discussed in Section 5.3.3.

In contrast, using various forms of fixed-point accumulation achieves accurate or near-accurate summation between inputs, making them the most common choice of addition implementation in recent SC works [SL17, HGT19, WLY20]. However, as we will show in the next subsection, they are expensive to implement since full adders required in APCs are larger, slower, and less energy efficient than OR gates, or multiplexers [LLR18].

### 5.1.3 Efficiency

Despite the precision issues outlined above, SC remains a promising candidate for accelerating various types of computation. The reason for the continued interest in it is the potential for unparalleled compute density [RLM20, WLY20]. Small, single-gate SC multipliers and adders can be orders of magnitude smaller than their fixed-point equivalents. However, this comes at the cost of increased computation latency, as SC operators require tens or hundreds of cycles depending on the stream length being used [RLM20, LLR18, WLY20]. To harness the most benefits out of SC, the area reduction offered by multipliers and adders needs to be used to enable higher spatial reuse, which in turn can amortize costs of conversion and memory accesses. When playing the spatial reuse game, the additional area and delay imposed by binary accumulation can wipe off a large part of the gains, as we will show shortly.

To demonstrate the difference between different styles of accumulation, we synthesized 256-wide dot products with different processing elements. The wide dot product is justifiable for modern neural networks that require multiply-accumulate computations with hundreds or thousands of inputs [SZ14, HZR16]. We compare fixed-point (*FXP*) multiply-accumulate computation and SC computation using accumulators (*Accumulator*), parallel counters (*Parallel Counter*), and OR gates (*OR*). We assume 6-bit input precision, which translates to 64 bits for the SC accumulators. Parallel counter and OR adder trees use split-unipolar computation [RLM20], which assumes one of the operands is bipolar while the other one can only be positive. The 64 bits are split into two 32-bit parts for the bipolar operand representing the positive and negative values separately. For positive values, the negative part is all zero, and vice versa. This is also a commonly encountered situation in neural networks employing the ReLU activation [SZ14, HZR16]. For SC multiplication, we assume that it is performed using simple AND gates. The accumulator configuration assumes 256 individual accumulators. We further compare non-saturating (*NS*) and saturating (*S*) adder trees for all configurations except the OR addition. The non-saturating configurations have adder width provisioned such that no truncation ever occurs. Saturating configurations limit the

width of all adders to 6 bits. We will further elaborate on the importance of saturation in Section 5.2. Results, synthesized using a commercial 28nm technology and Cadence Genus synthesis tool, are shown in Fig. 5.2.

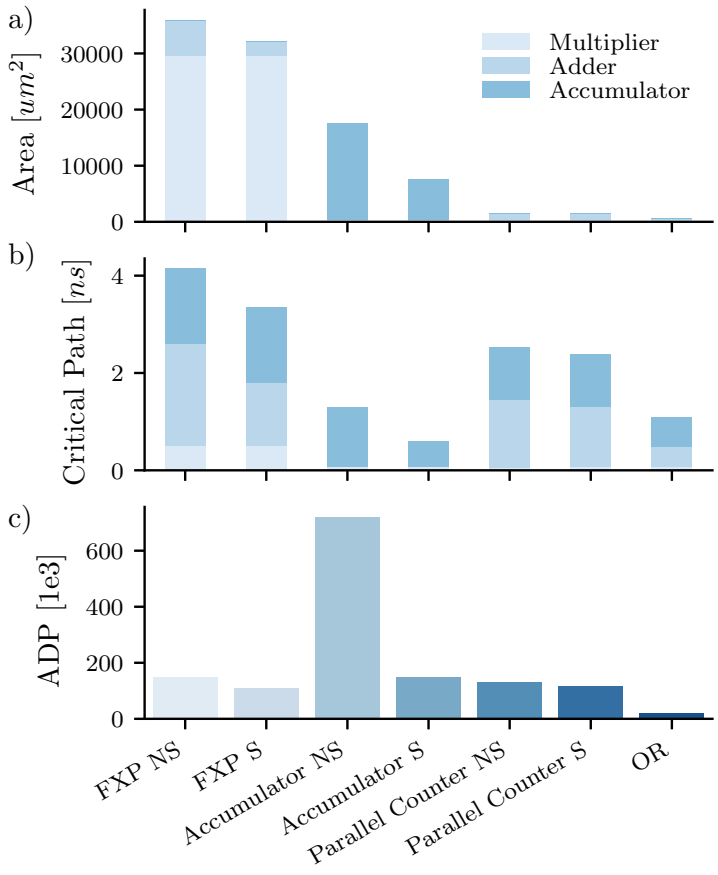



Figure 5.2: Area (a), critical path (b), and area-delay product (ADP) (c) for different dot product implementations. For SC implementations, ADP is calculated after multiplying the stream length, which is assumed to be 32 for each split-unipolar part.

Compared to the fixed-point baseline, SC offers significant area improvements. Non-saturating parallel counter implementation has 22.5X area improvement over the corresponding fixed-point design, while for a dot product with OR accumulation that advantage is 58.4X. The accumulator-based dot product can have a 2X area advantage, despite less efficient addition, because of the huge advantage provided by AND-based multipliers. This

explains why previous works have focused on optimizing SC multiplication since it is where most of the benefits of SC come from. However, the impact of addition cannot be ignored due to the SC's latency penalties. The non-saturating accumulator, parallel counter, and OR dot products have a 3.2X, 1.6X, and 3.8X critical path advantage over fixed-point. When accounting for stream length and combined with area in the form of an area-delay product, OR implementation has a 7X advantage, while the parallel-counter one has only 1.2X. The accumulator one has 5X worse throughput, showing that its use is not competitive without additional techniques reducing the stream length [SL17, HGT19]. The ADP difference between OR and parallel-counter dot products leads us to two important conclusions. First, *SC with binary accumulation has a fundamental limitation in possible performance improvement over fixed-point.* The small ADP margin can easily be whittled down when other system-level considerations come into play. Second, *there is a substantial performance gap between SC with binary and OR-based accumulation.* This *performance gap*, combined with the *precision gap* described in the previous section, *opens up an extensive design space of efficient and precise SC accumulation methods* that has not been given sufficient attention. In the next section, we will describe how this efficiency-precision gap can be bridged.

Finally, we want to acknowledge that the above analysis will depend on the dot-product size, precision, and stream length chosen, and different configurations might be more or less advantageous to certain implementations.

## 5.2 Range-extended OR Accumulation

### 5.2.1 Increasing the accuracy of OR accumulation

Given the importance of maintaining SC accumulation performance 5.1.3, we want to find a way to improve accumulation accuracy without sacrificing the benefits it offers. Between the three possible baseline implementations, MUX-based adders suffer from scaling on the inputs. Improving MUX accuracy requires increasing input stream length, which also affects

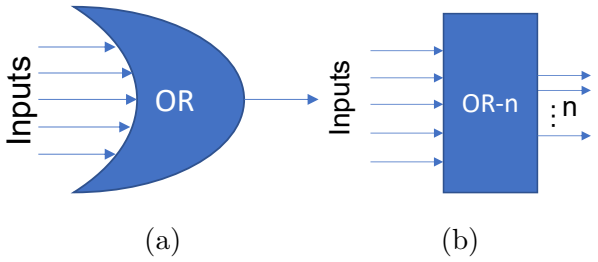



Figure 5.3: Overview comparison between (a) OR and (b) OR_n.

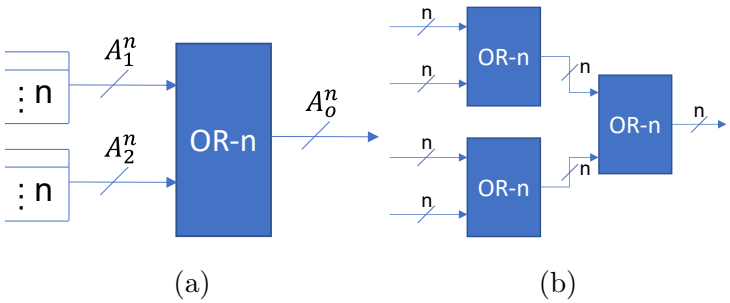



Figure 5.4: (a) Single OR_n gate and (b) cascaded OR_n gates.

the performance of multiplication and stream generation. Accumulator and counter-based adders achieve accurate addition and have performance issues instead of accuracy issues. OR-based adders suffer from the lack of output range. Adjusting the output range keeps the stream generation and multiplication components constant, and is thus the most plausible candidate for improvement. The accuracy deficit of OR accumulation comes from the bitwise computation nature. The outputs of an OR accumulation have the same stream length as the inputs. Fixed-point accumulation adds all input bits together. It is equivalent to concatenating the input bit streams together, resulting in much longer effective stream length at the output. Therefore, to increase the output precision of an OR gate, we want to increase the number of output bits corresponding to each input bit to $n$. We denote the resulting range-extended OR gate as OR_n and a regular OR gate can be seen as a special case where $n = 1$. A comparison of an OR_n gate and a regular OR gate is shown in Fig. 5.3. In theory, any logic operation that takes 1 bit from each input and outputs n bits is a valid OR_n gate,

leading to $2^{512}$ unique truth tables exist for OR_2 with 256 inputs (256 input bits, 2 output bits). We will discuss how we determine the OR_2 gate, and then extend the derivation to larger OR_n.

Given the large number of possible OR_2 configurations, we limit our search space through the following steps:

1. Limit the search to two-input OR_n gates. Fig.5.4a illustrates the idea of a 2-input OR_n gate. $A^n$ represents n bits, and this notation will be used in later discussions to represent $n$ input or output bits. A two-input OR_n gate takes two $A^n$ inputs and creates an $A^n$ output. Larger OR_n gates can be built by cascading multiple levels of two-input OR_n gates, as shown in Fig. 5.4b). Focusing on two-input gates ensures that the cost of large accumulations scale linearly with the size of accumulation, and reduces the search space for OR_n to $2^{n2^{2n}}$ (2n input bits, n output bits), or $2^{32}$ for OR_2.

2. Focus on the transfer function between the sum of input bits and the sum of output bits. In other words, instead of trying to find the function $A_o^n = f(A_1^n, A_2^n)$, we try to find the sum transfer function $\sum(A_o^n) = f_s(\sum(A_1^n) + \sum(A_2^n))$. The exact form of the function will be determined later after considering other constraints. This concept is illustrated in Tab. 5.2, which depicts one possible implementation of OR_2. The sum transfer function loses positional information of the individual input bits and instead forces the computation to be associative. In other words, the order of the input bits does not affect the sum of the output bits, and the output bits can be adjusted as long as the sum does not change. As discussed in Sec. 5.2.3, the associative constraint allows for more efficient training of models using OR_n accumulation. The sum transfer function constraint reduces the search space to $(n+1)^{2n+1}$ (n+1 output sum values from n output bits, 2n+1 input sum values from 2n input bits), or $3^5$ for OR_2.

3. The sum transfer function should be non-decreasing, and should use all possible sum

values of the n output bits. The former ensures that OR_n behaves similarly to normal accumulation, which simplifies training. The latter ensures that the output bits are fully utilized. These two constraints limit the search space to $\binom{2n}{n}$ (n increments in 2n entries), or 6 for OR_2.

Table 5.2: Comparison between a full truth table and a sum transfer function. This is one possible truth table of a two-input OR_2 gate that corresponds to Candidate 6 in Tab. 5.3. This particular truth table sets the two output bits to $e = a + c + bd$ and $f = b + d + ac$ respectively, assuming $a$, $b$, $c$, and $d$ are the four input bits, and $e$ and $f$ are the two output bits.

| Input Bits | | | | Input Sum | Output Bits | | Output Sum |
|---|---|---|---|---|---|---|---|
| $a$ | $b$ | $c$ | $d$ | | $e$ | $f$ | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 |
| 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 |
| 0 | 0 | 1 | 1 | 2 | 1 | 1 | 2 |
| 0 | 1 | 0 | 0 | 1 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 | 2 | 1 | 1 | 2 |
| 0 | 1 | 1 | 0 | 2 | 1 | 1 | 2 |
| 0 | 1 | 1 | 1 | 3 | 1 | 1 | 2 |
| 1 | 0 | 0 | 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 2 | 1 | 1 | 2 |
| 1 | 0 | 1 | 0 | 2 | 1 | 1 | 2 |
| 1 | 0 | 1 | 1 | 3 | 1 | 1 | 2 |
| 1 | 1 | 0 | 0 | 2 | 1 | 1 | 2 |
| 1 | 1 | 0 | 1 | 3 | 1 | 1 | 2 |
| 1 | 1 | 1 | 1 | 4 | 1 | 1 | 2 |

Tab. 5.3 shows the 6 candidate transfer functions for the OR_2 gates that satisfy all the constraints.

To find the best among the 6 candidate transfer functions, we examine their behav-

Table 5.3: Candidate transfer functions between input and output sums for 2-input OR_2.

| Candidates | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| Input Sum | Output Sum | | | | | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 1 | 1 | 1 |
| 2 | 0 | 1 | 1 | 1 | 1 | 2 |
| 3 | 1 | 1 | 2 | 1 | 2 | 2 |
| 4 | 2 | 2 | 2 | 2 | 2 | 2 |

iors when expanded to 3 OR_2 inputs shown in Tab. 5.4. Of the 6 candidates, half of them lose the associative property, and the output depends on the position of the input. Take Candidate 2 as an example. $OR\_2\_2(OR\_2\_2(\{0,1\},\{0,1\}),\{0,0\}) = \{0,0\}$ whereas $OR\_2\_2(OR\_2\_2(\{0,0\},\{0,0\}),\{1,1\}) = \{0,1\}$, even though the input bits add up to 2 for both cases. For the rest of the three candidates, candidate 1 only activates when almost all input bits are 1, which is unlikely for neural networks with very wide accumulations. Candidate 4 faces a similar issue where the second bit only activates when all input bits are 1. We expect OR_2 using candidate 1 or 4 to perform poorly, and this is confirmed in Tab. 5.5. For the same 4-layer CNN (TinyConv) used in [LSC18], candidate 1 does not converge, candidate 4 barely offers any benefit over OR_1, and only candidate 6 is a noticeable improvement over OR_1. As a result, we choose candidate 6 as the sum transfer function of OR_2.

The transfer function of OR_2 can be written as

$$\sum(A_o^2) = \begin{cases} \sum(A_1^2) + \sum(A_2^2), & \sum(A_1^2) + \sum(A_2^2) < 2 \\ 2, & \text{otherwise} \end{cases} \tag{5.1}$$

where $A_o^2$ is the output tuple and $A_1^2$ and $A_2^2$ are the input tuples. Larger OR_n can be derived similarly by replacing 2 in Eq. 5.1 with $n$, as shown in Eq. 5.2. Alternatively, OR_n

Table 5.4: Candidate transfer functions between input and output sums for 3-input OR_2.

| Candidates | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| Input Sum | Output Sum | | | | | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 1 | 1 | 1 |
| 2 | 0 | 0,1 | 0,1 | 1 | 1 | 2 |
| 3 | 0 | 0,1 | 1 | 1 | 1,2 | 2 |
| 4 | 0 | 1 | 1,2 | 1 | 1,2 | 2 |
| 5 | 1 | 1 | 2 | 1 | 2 | 2 |
| 6 | 2 | 2 | 2 | 2 | 2 | 2 |

can also be viewed as a saturating adder that saturates at $n$. A key feature that sets OR_n apart from other designs with a saturating adder/accumulator is that it saturates after every addition and operates only on unipolar inputs. The former ensures that an OR_n gate does not introduce additional bits, as an OR_2 gate will always generate 2 bits and an OR_3 gate will always generate 2/3 bits, depending on the implementation. This linear scaling with the size of accumulation is advantageous. The latter feature ensures that saturation is well-defined. Hardware implementations saturate the outputs after each OR_n accumulation, minimizing the number of wires out of an OR_n gate and allowing the same OR_n gate to be reused for larger accumulations. In software modeling of OR_n, saturation can be delayed to the end to better utilize dot-product instructions present in recent CPUs and GPUs while still being functionally identical to the hardware version. A bipolar saturating adder, on the other hand, lacks this convenience. For example, if we consider a saturating addition that saturates at {5,-5} between 3, 4, and -5, saturating after adding 3 and 4 will produce a different result compared to saturating after adding all three values together. The accumulation result from a bipolar saturating adder will depend on the order of computation

and will be more difficult to model, particularly if saturation occurs frequently.

$$\sum(A_o^n) = \begin{cases} \sum(A_1^n) + \sum(A_2^n), & \sum(A_1^n) + \sum(A_2^n) < n \\ n, & \text{otherwise} \end{cases} \tag{5.2}$$

Table 5.5: TinyConv Neural network accuracy of Candidate 1, 4, and 6 for OR_2. Models are trained using 32-bit streams on CIFAR-10.

| Candidate | OR_1 | 1 | 4 | 6 |
|---|---|---|---|---|
| Accuracy | 74.81% | 10% | 75.31% | 78.92% |

Despite focusing on the ease of implementation when deciding between candidates, the final OR_n offers lower error compared to OR when normalized to the same total output stream length. This means using $k$ input bits for OR_n and $kn$ input bits for OR, resulting in $kn$ output bits for both cases. Assuming different bits in a stream have independent and identical distribution (IID), the output value follows a binomial distribution, with root-mean-squared (RMS) error being $\sqrt{\frac{v(1-v)}{kn}}$, where $v$ is the expected error and $kn$ is the stream length. This is a concave function over $v$ [3]. Due to the concavity of the error function, having different probability values in different parts of the stream reduces the final average error. For OR_n, each group of $k$ output bits represents a different value. As a result, we expect even OR_2 to outperform OR in accuracy when using half the input stream length.

### 5.2.2 Efficient OR_n implementation

While Section 5.2.1 defines the optimal transfer function between input and output sums for OR_n, the transfer function does not uniquely define the truth table. Take OR_2, for instance. Given two $A^2$ inputs 00 and 01, the transfer function determines that the $A^2$

[3]This error function assumes using a true random number generator (TRNG) for generation. Using a more accurate generator reduces overall error, but the error is still a concave function with respect to the expected output value

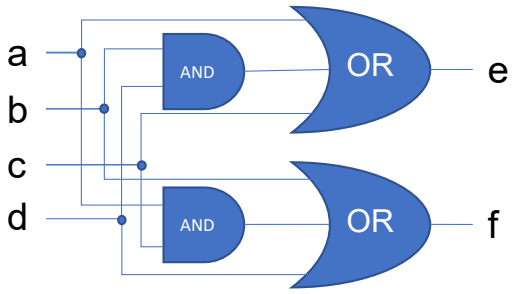



Figure 5.5: OR_2 circuit implementation. Similar to Tab. 5.2, a, b, c, and d are the four input bits, and e and f are the two output bits.

outputs sum up to 1 but does not decide whether the outputs should be 01 or 10. For OR_2, four output positions can be either 01 or 10, resulting in 16 possible truth tables similar to the Tab. 5.2. Fig.5.5 shows a possible circuit implementation of OR_2 with the truth table in Tab. 5.2. We synthesized 256-wide adder trees for all 16 truth tables, and the area-delay product comparison is shown in Fig. 5.6. Configuration 4 is the best-performing option, and we used that for all performance evaluations of OR_2. It is also 3.3X better in terms of ADP compared to non-saturating parallel-counter-based addition.

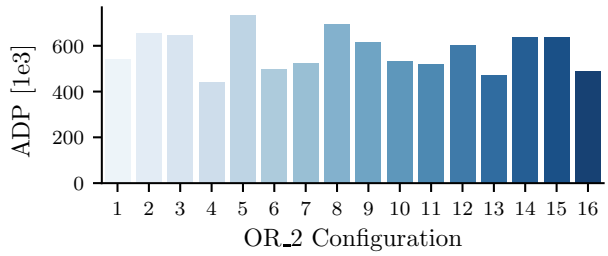



Figure 5.6: ADP comparison between different OR_2 truth table implementations.

OR_3 uses a saturating adder instead of the parallel implementation of OR_2. From Eq. 5.2, OR_n behaves like a saturating adder that saturates at $n$. Instead of adding the three parallel output bits after accumulation, every three input bits are added together using a full adder, and 2-bit saturating adders perform the rest of the accumulation. The two implementations are illustrated in Fig. 5.7. We synthesized both options, and the

saturating adder version has ADP 2X lower than the parallel implementation, so we use the saturating adder version for performance evaluations of OR_3. Its ADP improvement over parallel-counter addition is 1.6X. Because of that, we do not explore $n > 3$, since we do not expect it to have better performance than fixed-point accumulation.

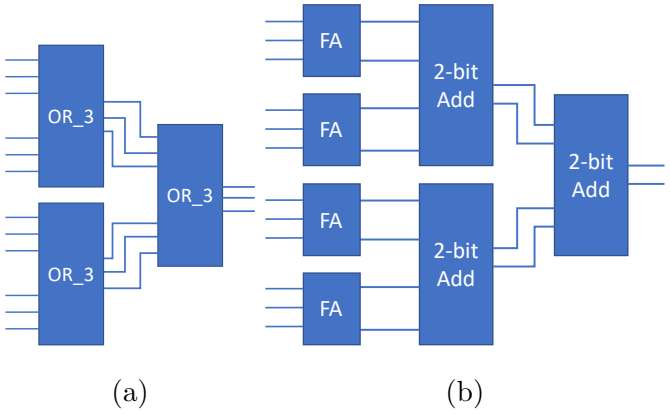



Figure 5.7: (a) Parallel and (b) saturating adder implementation of OR_3.

### 5.2.3 Efficient training for OR_n accumulation

While we focus on improving inference performance with SC and OR_n, we need to ensure that models are trained so that they work well for OR_n SC during inference. Training is an important aspect of deep learning, and modifications to the compute algorithm during inference need to be easily trainable on commodity hardware. Our training setup is based on the idea of straight-through estimators (STE) [Ben13]. Fig. 5.12a shows the basic STE setup for training inaccurate neurons. The forward pass accurately models the inaccurate computation (in our case SC stream computation), while the backward pass ignores the inaccuracies. For REX-SC, we use an efficient simulation framework to simulate SC-based computation in the forward pass. This ensures that the training results are representative of what we will get on SC hardware. Every 32 bits in an SC bitstream is packed into a 32-bit integer to maximize performance. The simulation framework is written in CUDA C++[NVF21] to maximize performance on Nvidia GPUs and utilizes AVX2 intrinsics[Int21]

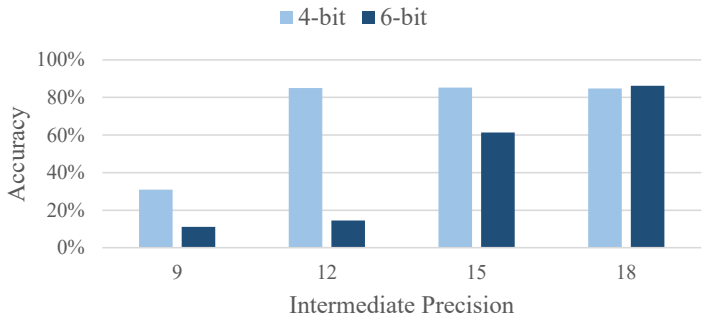



Figure 5.8: TinyConv accuracy of with fixed-point computation using different intermediate precision.

with OpenMP [DM98] to maximize performance on x86 CPUs. WMMA functions[NVI21] are used for Nvidia GPUs with Compute Capability $\geq 8.0$.

Backpropagation uses floating-point computation during training, and the non-linearity of OR_n accumulation requires modeling in the backward pass. To understand the need for additional modeling during the backward pass, we need to consider the saturation behavior of OR_n. As OR_n saturates at n, output values close to n should result in lower gradient values on the corresponding inputs and weights. The vanilla STE setup ignores the saturation behavior during the backward pass and thus degrades the accuracy of the final trained model. This is an issue even for fixed-point quantization with limited accumulation precision, as shown in Fig. 5.8. When there is not enough intermediate precision and the accumulation saturates, accuracy drops significantly when the saturation is not modeled. [RLM20] shows that it is possible to model OR accumulation as normal accumulation + activation function. This allows the usage of optimized convolution and linear kernels in popular deep learning frameworks. OR_n modeling uses a similar concept to modeling OR_1, so we will go through the derivations for OR_1, and then extend the idea to OR_n. For an OR_1 gate with inputs $a_k$, the expected output value is equal to

$$OR\_1 = 1 - prob(0\ input\ is\ 1) = 1 - \prod(1 - a_k) \tag{5.3}$$

By assuming all inputs are the same, Eq. 5.3 simplifies to $1-(1-s/n)^n$, where s is the normal sum of the inputs. When n is large, it further simplifies to

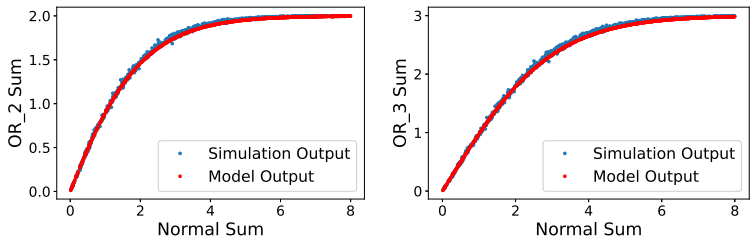
$$OR\_1 \approx 1 - e^{-s} \tag{5.4}$$

Similar concepts can be applied to OR_2. The OR_2 output can then be approximated as

$$OR\_2(a_k) \approx 2 - 2e^{-s} - se^{-s} \tag{5.5}$$

The same idea can be extended to higher-degree OR_n, which is shown in Eq. 5.6.

$$OR\_n(a_k) \approx n - \sum_{i=0}^{n-1} \frac{(n-i)s^i}{i!} e^{-s} \tag{5.6}$$




(a) (b)

Figure 5.9: Modeling performance of (a) OR_2 and (b) OR_3.

The backpropagation through a neural network layer using OR_n accumulation is modeled as a normal convolution/linear backward pass (layer_bwd) with an added pointwise function, as is shown in Eq. 5.7 and 5.8, where $g_o$ is the output gradient.

$$OR\_n\_layer\_bwd(g_o) = layer\_bwd(OR\_n\_bwd(g_o)) \tag{5.7}$$

$$OR\_n\_bwd(g_o) = g_o * (1 + \sum_{i=0}^{n-1} \frac{s^i}{i!} e^{-s}) \tag{5.8}$$

Fig. 5.9 shows the modeling performance of OR_n approximations. 100 inputs are used in the accumulation, each having a stream length of 1024. Our proposed approximation correctly captures the trend of OR_n accumulation. Since the approximation is only a

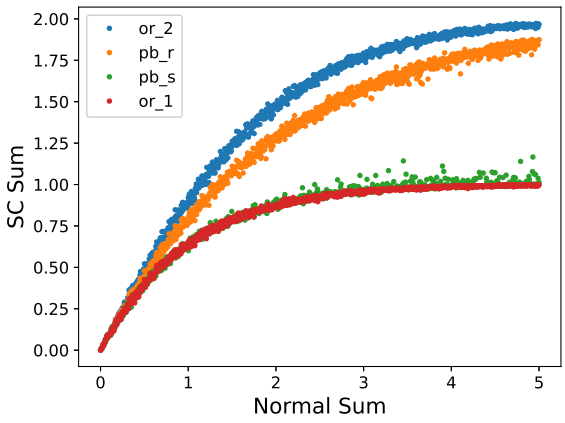
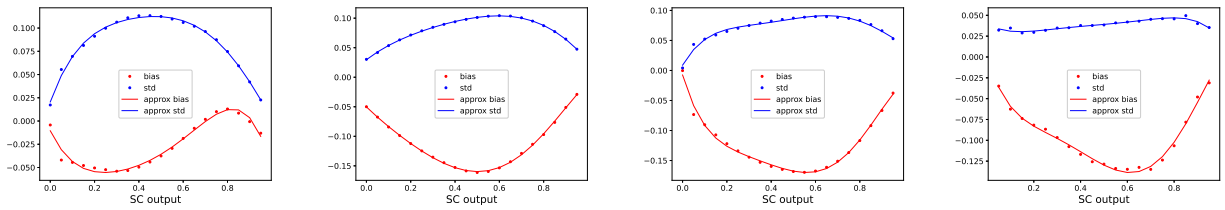



Figure 5.10: Modeling performance comparison of OR_2 and 2-way partial binary accumulation. pb_r and pb_s use the same inputs and only the order of computation is changed, such that for pb_r partial sums are roughly equal, and for pb_s they are very different.

function of normal accumulation sum S, it can also be modeled as a single activation function and is efficient to implement during training. Such models are only possible when the OR_n function satisfies the associative property mentioned in Sec. 5.2.1. Consider partial binary accumulation [LRP21] (PB) which also tries to extend the range of SC accumulation. In PB, large accumulations are broken up into multiple groups. Each group uses OR gates for accumulation, and results from different groups are added together using binary adders. Using two kinds of accumulators make PB non-associative, which is demonstrated in Fig. 5.10. Compared to OR_2 which maintains its accumulation behavior regardless of the order of computation, PB has different behaviors depending on how the accumulation is broken up. The output response is very different when the two groups are roughly equal (pb_r) and when the two groups are very different (pb_s). While pb_r behaves similarly to OR_2, pb_s barely offers any benefit over OR_1. Given the randomness of neural networks, it is impossible to predict how balanced the groups are, and thus impossible to use a single activation function. This will have a profound impact on training performance, as we will show in Sec.5.3.3.

While the stream simulation improvements for the forward pass reduce forward propaga-




(a) Layer 1 (b) Layer 2 (c) Layer 3 (d) Layer 4

Figure 5.11: Difference between outputs from stream computation and normal computation+activation.

tion time, it is still expensive compared to a normal forward pass using single/half-precision floating-point numbers. To further reduce the training overhead for REX-SC, we propose to replace stream simulation with activation calibration and error injection (a+e for short) for most of the training epochs. Since the activation approximation is sufficient for the backward pass, one may ask why it is not sufficient for the forward pass. The main issue is that the activation functions cannot completely capture the characteristics of OR_n accumulation. Fig. 5.11 shows the mean and standard deviation of the difference between actual stream output and activation output for the same 4-layer CNN used in Tab. 5.5. The average error (denoted as bias in the figure) is non-zero, implying that the activation function is not perfect. Since the average error is different for each layer, it is impossible to use a single activation function for all layers. The standard deviation of the error (denoted as std in the figure) means it is impossible to capture all the properties of OR_n accumulation with only the activation function. Fortunately, both the mean and variance of the error are smooth functions of the activation value. We can thus approximate the two parts of the error with polynomial functions using standard linear least squares regression [Wat67], as is shown in the curves. The fitted mean curve is then added to the post-activation value as a calibration step for the activation, and the variance curve is used to inject random error to each activation with variance equal to the value predicted by the curve. Fig. 5.12 compares the training setup of stream and a+e training steps. While the a+e step seems more complicated, the

additional operators are all point-wise operations and can be fused to reduce overhead. The step to update the error curves for a+e is performed only a few times per epoch of training to amortize its relatively large cost.

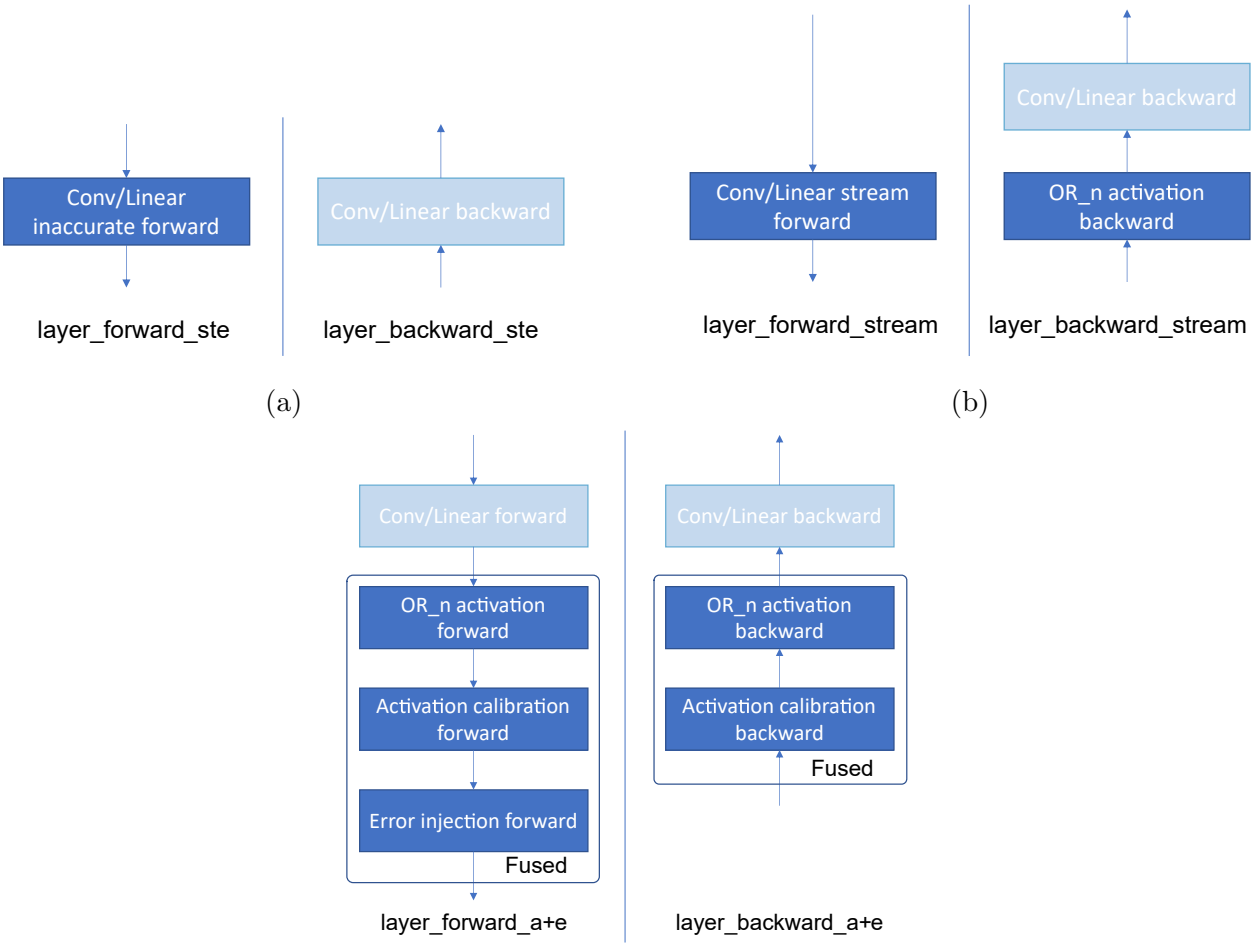



Figure 5.12: Comparison between (a) vanilla STE [Ben13], (b) stream with OR_n activation and (c) a+e training step.

With the optimized stream computation simulation, backward propagation using activation function, and a+e step, the overall training setup is shown in Alg. 1. Models are first trained using the a+e training step and then fine-tuned with stream training for a few epochs.

**Algorithm 1** Training setup for OR_n.

**Require:** Epochs with a+e training $n_e$, epochs with stream training $n_s$, number of training steps between a+e update $k_e$, total number of training mini batches $k_0$, dataset $d$.

```
n ← 0
while n < n_e do                                              ▷ a+e training phase
    k ← 0
    while k < k_0 do
        x ← d[k]
        if k mod k_e = k_e − 1 then                           ▷ update a+e parameters
            for all layers do
                x_l ← layer input
                y_{a+e} ← layer_forward_a+e(x)
                y_{stream} ← layer_forward_stream(x)
                Update layer mean and variance curves
            end for
        else
            x ← model input
            y = model_forward_a+e(x)
        end if
        model_backward_a+e(y, model target)
        k ← k + 1
    end while
    n ← n + 1
end while
while n < n_e + n_s do                                        ▷ stream training phase
    k ← 0
    while k < k_0 do
        x ← d[k]
        y ← model_forward_stream(x)
        model_backward_a+e(y, model target)
        k ← k + 1
    end while
    n ← n + 1
end while
```

## 5.3 Results

### 5.3.1 Evaluation

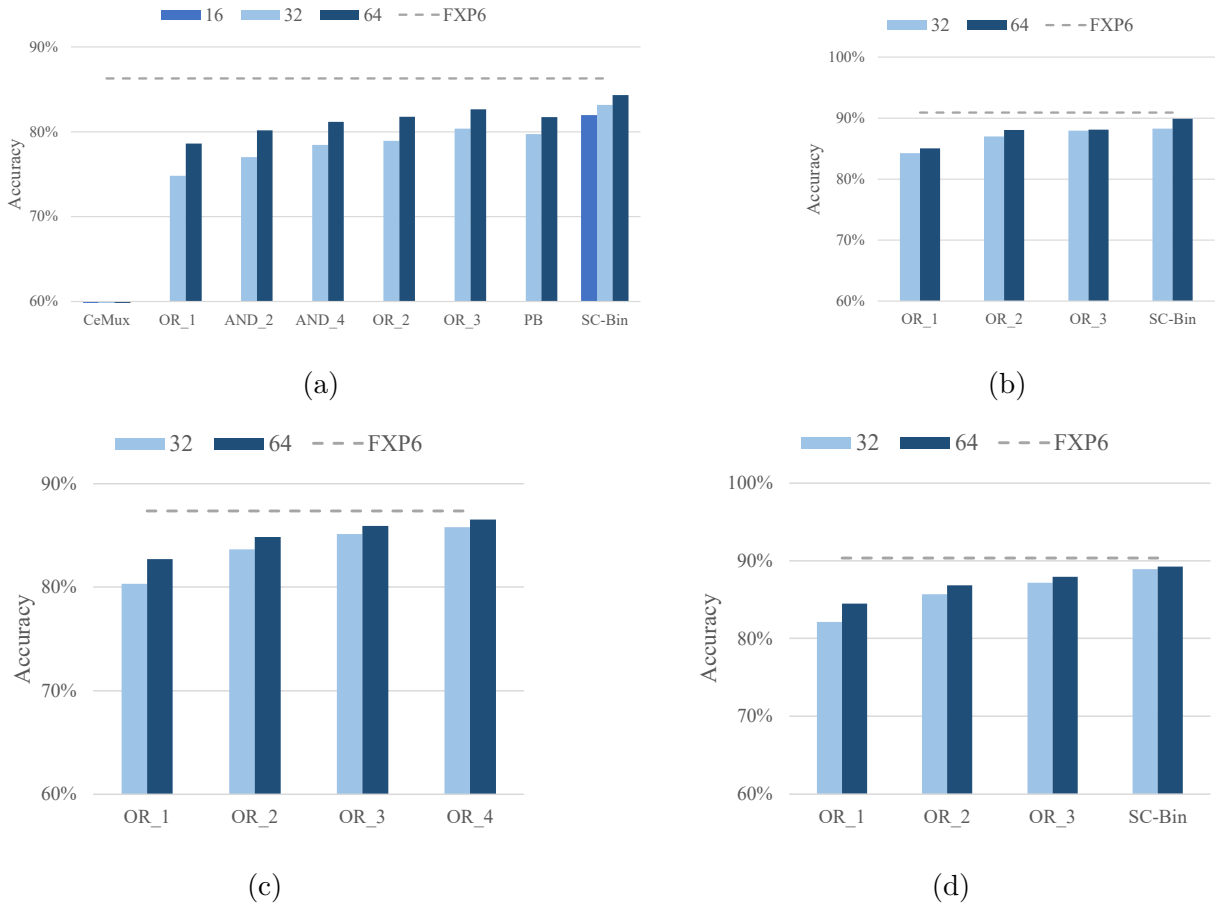



Figure 5.13: Accuracy comparison for (a) TinyConv and (b) VGG-16 on CIFAR-10, (c) Resnet-18, and (d) Resnet-34 on ImageNet. 16-bit SC-Bin only has an accuracy result for TinyConv as it does not converge for other larger models.

To measure the accuracy benefits of extended-range SC, we evaluate accuracy on image classification datasets CIFAR-10 and ImageNet [KSH12]. For CIFAR-10, we use a 4-layer convolutional neural network (TinyConv) [LSC18] and VGG-16 [SZ14]. For ImageNet, we use Resnet-18 and Resnet-34 [HZR16]. Models are trained using PyTorch 1.12.0. For a+e,

the error curves are adjusted 5 times per epoch. A lower calibration frequency is possible but does not yield additional performance benefits as the calibration time is sufficiently amortized. CIFAR-10 models are trained for 100 epochs, where 80 epoch use a+e and 20 use stream simulation. ImageNet models are trained for 70 epochs, where 65 epochs use a+e and 5 use stream simulation. Models are trained using maximal-length LFSRs as the RNG and use the same RNG sharing scheme proposed in [LRP21]. LFSR seeds are shared between different filters and between different input batches. The sharing scheme is demonstrated in Tab. 5.6 for a layer with 4x4 input and 3x3 weight and a stream length of 8 (3-bit LFSR). Seeds are also shared within the same input and weight filter due to the limited number of unique seeds for a maximal-length LFSR ($2^n - 1$ for an n-bit LFSR) and are correctly modeled. Streams are converted to fixed-point numbers using a counter between layers. The training code is available at `https://github.com/nanocad-lab/rex-sc`.

Table 5.6: LFSR seed sharing scheme demonstration.

| Position | Seed values |
|---|---|
| Input 1 | 0,1,2,3,4,5,6,0,1,2,3,4,5,6,0,1 |
| Input 2 | 0,1,2,3,4,5,6,0,1,2,3,4,5,6,0,1 |
| Filter 1 | 6,0,1,2,3,4,5,6,0 |
| Filter 2 | 6,0,1,2,3,4,5,6,0 |

### 5.3.2 Accuracy improvements

Accuracy comparisons between OR_n and other alternatives are shown in Fig. 5.13a for TinyConv [LSC18] on CIFAR10. OR_1 uses OR gates for accumulation similar to the setup used in [RLM20]. SC-Bin uses parallel counters for accumulation, similar to the setup in [SL17]. It can also be seen as an upper bound of accuracy for uSADD and uNSADD proposed in [WLY20], as the latter two also try to reduce multiplication error while sacrificing some

accumulation accuracy in the pursuit of streaming compute. MUX-based additions suffer from scaling issues on the weight values, which prevents convergence even with the improvements from CeMux[BH22]. PB uses partial binary accumulation setup used in [LRP21]. Accuracy of OR_2 is comparable to PB at both stream lengths, while OR_3 outperforms both. SC-Bin has a 2-4% point accuracy advantage over OR_2 and a 1-3% point advantage over OR_3. Compared to the FXP6 baseline using 6-bit fixed-point multiply-accumulate, OR_2 has a 4-6% point accuracy deficit, while OR_3 narrows the gap to 3-5% points. Fig. 5.13a also shows the results of the same concept applied to AND multiplication, denoted as "AND_2" and "AND_4". AND_n takes groups of n bits from each of the two multiplicands and multiplies all bits from the first group with all bits from the second group. Traditional AND multiplication can thus be seen as AND_1. AND_2 and AND_4 increase multiplication stream length by 2X and 4X respectively and increase overall area and energy by the roughly same amount. While AND_n can also improve accuracy, it is not energy- or area-efficient. AND_4 roughly matches OR_1 when using half the stream length, but consumes $\approx 2X$ the area and energy.

Table 5.7: Array size, compute area and power, clock period, inference latency, energy, and energy improvement compared to 6-bit fixed-point, for different models and datasets with different types of SC accumulation.

| | | | | | | CIFAR-10 TinyConv | | | CIFAR-10 VGG | | | ImageNet ResNet-18 | | | ImageNet ResNet-34 | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Architecture | N | Area | Period | Power | Stream | Latency | Energy | E. Impr. | Latency | Energy | E. Impr. | Latency | Energy | E. Impr. | Latency | Energy | E. Impr |
| | | [mm2] | [ns] | [mW] | Length | [us] | [uJ] | vs FXP6 | [us] | [uJ] | vs FXP6 | [us] | [uJ] | vs FXP6 | [us] | [uJ] | vs FXP6 |
| FXP6 | 9 | 2.97 | 4.5 | 692 | | 4.5 | 7.9 | 1.0 | 76.5 | 132.2 | 1.0 | 446.8 | 772.8 | 1.0 | 817.5 | 1413.2 | 1.0 |
| SC_Bin | 41 | 3.12 | 1.6 | 3282 | 32 | 2.6 | 12.3 | 0.6 | 44.6 | 207.1 | 0.6 | 246.0 | 1145.4 | 0.7 | 449.8 | 1666.0 | 0.8 |
| | | | | | 64 | 5.3 | 21.0 | 0.4 | 89.2 | 353.5 | 0.4 | 492.1 | 1953.0 | 0.4 | 899.6 | 3142.4 | 0.4 |
| SC uSADD | 12 | 2.93 | 1.1 | 970 | 32 | 20.6 | 31.5 | 0.2 | 347.8 | 530.3 | 0.2 | 2032.3 | 3101.2 | 0.2 | 3718.2 | 4243.8 | 0.3 |
| | | | | | 64 | 41.3 | 51.6 | 0.2 | 695.7 | 867.7 | 0.2 | 4064.6 | 5072.5 | 0.2 | 7436.5 | 7850.5 | 0.2 |
| SC uNSADD | 12 | 2.95 | 1.2 | 1026 | 32 | 21.8 | 33.9 | 0.2 | 368.2 | 570.7 | 0.2 | 2151.3 | 3337.1 | 0.2 | 3936.0 | 4675.4 | 0.3 |
| | | | | | 64 | 43.7 | 56.3 | 0.1 | 736.4 | 948.5 | 0.1 | 4302.6 | 5544.4 | 0.1 | 7872.0 | 8713.7 | 0.2 |
| SC OR_1 | 64 | 3.21 | 2.4 | 488 | 32 | 1.5 | 1.0 | 8.0 | 27.4 | 17.8 | 7.4 | 155.9 | 101.4 | 7.6 | 285.5 | 267.2 | 5.3 |
| | | | | | 64 | 3.0 | 1.7 | 4.6 | 54.8 | 31.1 | 4.2 | 311.7 | 177.4 | 4.4 | 571.0 | 406.4 | 3.5 |
| SC OR_2 | 51 | 3.2 | 2.2 | 596 | 32 | 2.7 | 2.1 | 3.7 | 43.2 | 34.8 | 3.8 | 227.8 | 183.8 | 4.2 | 417.8 | 404.7 | 3.5 |
| | | | | | 64 | 5.3 | 3.7 | 2.1 | 86.4 | 60.5 | 2.2 | 455.6 | 319.6 | 2.4 | 835.5 | 653.6 | 2.2 |
| SC OR_3 | 50 | 3.12 | 2.2 | 650 | 32 | 2.6 | 2.3 | 3.4 | 42.4 | 37.3 | 3.5 | 231.6 | 204.5 | 3.8 | 424.1 | 437.0 | 3.2 |
| | | | | | 64 | 5.2 | 4.0 | 2.0 | 84.7 | 64.9 | 2.0 | 463.1 | 355.1 | 2.2 | 848.1 | 712.8 | 2.0 |

Accuracy results of VGG-16 on CIFAR-10 and Resnet-18/34 on ImageNet are shown in Fig. 5.13b) and 5.13c)/5.13d). Both OR_2 and OR_3 achieve similar accuracy to 6-bit fixed-point on VGG-16. While OR_2 and OR_3 improve Top-5 accuracy by 3-5% points compared to OR_1 on ImageNet, they are 2-5% points lower than FXP6. The accuracy of OR_N is likely limited by the training hyperparameters. For instance, doubling the number of epochs from 35 to 70 improves accuracies by 1.5-2% points for OR_n, and longer training times should improve accuracies even further. Compared SC-Bin that has unlimited accumulation precision, OR_2 and 3 reduces the accuracy gap while requiring at most 2 bits of accumulation precision. Compared to fixed-point computation and SC-Bin, OR_N enables scaling to higher performance levels. While SC-Bin using 16-bit streams has accuracy between 32-bit and 64-bit OR_3 on TinyConv, it is not a useful setup for the other models. Both fixed-point and SC-Bin have convergence problems in VGG and Resnets when dropping precision further, as weight values tend to underflow the smallest representable value with 5-bit fixed-range quantization (16-bit stream for SC).

Table 5.8: Iteration time comparisons between OR_n and partial binary accumulation. Time is measured in s/256 images on an RTX 3090. The numbers before "PB" are the sizes of OR accumulation, with smaller numbers leading to more binary accumulations.

| | TinyConv (CIFAR-10) | | Resnet-18 (ImageNet) | | |
|---|---|---|---|---|---|
| Stream Length | 5x8 PB | OR_n | 5x8 PB | OR_n | a+e |
| 32 | 0.092 | 0.041 | 8.08 | 1.09 | 0.26 |
| 64 | 0.108 | 0.052 | 9.24 | 1.64 | 0.26 |
| 128 | 0.135 | 0.081 | 11.61 | 2.72 | 0.26 |

### 5.3.3 Training speed improvements

While partial binary accumulation and OR_2/3 have similar accuracy benefits over OR_1 as shown in Section 5.3.2, partial binary accumulation complicates the training process of neural networks. In contrast to OR_2 and OR_3, it is not feasible to approximate partial

binary accumulation with a single activation function, since partial binary accumulation is not associative by design.

Iteration speed comparisons between OR_n and partial binary accumulation are shown in Tab. 5.8. Different n values for OR_n only change the activation function used during training and do not significantly affect training time. For smaller models like TinyConv where the memory requirement is not an issue, PB is between 1.5X and 2.2X slower. Larger models like Resnet-18 suffer more due to memory size limitations. Higher levels of binary accumulation exacerbate this effect, making PB up to 7.4X slower than OR_n to train. a+e further reduces iteration time by at least 4X. Since the runtime of a+e is not dependent on stream length, it is especially useful when training for longer streams.

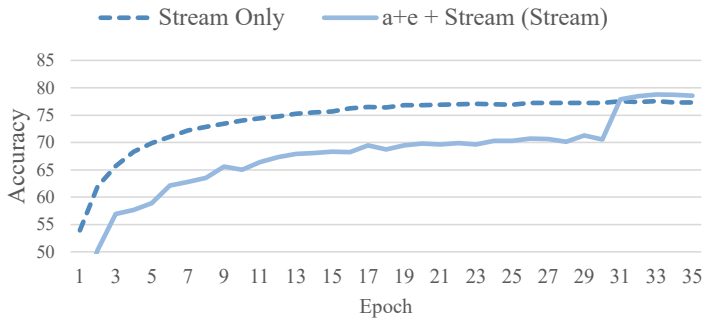



Figure 5.14: Convergence behavior of stream only and a+e. The drop in accuracy at epoch 31 for a+e is when switching to using streams for the forward pass.

Fig. 5.14 compares the convergence behaviors of training with stream only and with a+e replacing most of the epochs. Training is limited to 35 epochs for both cases, as stream training with the full 70 epochs takes too long. a+e has similar convergence behavior as stream only and achieves similar accuracy in the end. Overall training time is reduced by at least 3X with a+e compared to stream only and at least 22X compared to training with PB as shown in Tab. 5.9.

Table 5.9: End-to-end training time comparison in hours between OR_n and partial binary accumulation. Results for OR_n and PB are estimated as they take prohibitively long to train.

| Stream Length | 5x8 PB | OR_n | a+e |
|---|---|---|---|
| 32 | 710.9 | 95.9 | 31.6 |
| 64 | 812.4 | 144.2 | 35.0 |

### 5.3.4 Performance

To evaluate the performance benefits of using OR_n implementations, we use the previously synthesized adder and dot product results together with buffers and SNG results using commercial 28nm technology and Cadence Genus synthesis tools. SNGs are based on maximum-length LFSRs, shared across different processing elements, and comparators. We use 6-bit fixed-point (FXP6) as a baseline and compare its performance with SC implementations. Besides, SC with binary accumulation (SC_Bin), OR_1, OR_2, and OR_3, we include uSADD and uNSADD (scaled and non-scaled adders) from [WLY20]. Both uSADD and uNSADD use uMUL multipliers, which offer high accuracy without retraining, but at a significant hardware cost. To evaluate system-level performance, we assume dot-product processing elements are organized as a N-by-N GEMM array, similar to recently proposed SC accelerators [CVR14, WLY20, RLG22]. We assume the individual PE to have a width of 256. We use a $N = 64$ array using OR-based SC dot products as a baseline, which occupies an area of $3.2mm^2$, and we size up all other configurations to be as close to this area budget as possible. We then try to normalize the throughput w.r.t. FXP6 for all configurations, assuming 64-long streams, by increasing or lowering clock frequency, and corresponding power to take advantage of voltage-frequency scaling. SC with binary accumulation and OR_1 accumulation cannot be fully throughput normalized to FXP6 due to impossibly high and low resulting voltage requirements, respectively. As a result of throughput normalization, OR_n configurations can use lower clock frequency and consume lower power than other SC

configurations with a similar area. We use an analytical model that takes the array size and layer parameters, such as input and filter size, number of filters etc., and calculates both the number of compute iterations as well as memory accesses required to process a given layer, assuming output stationary dataflow. We focus on convolutional layers, as they dominate runtime and energy for all explored models. Our model assumes the memory bandwidth is provisioned such that computation is never stalled and that computation has maximum valid utilization. We use the clock period, stream length, and iteration count to estimate the latency of each model's inference. We estimate inference energy using the latency, synthesized design power, memory access count, and energy obtained using the CACTI 6.5 tool [MBJ09].

Performance results including improvement over 6-bit fixed-point are shown in Tab. 5.7. First, we show that SC with parallel counter accumulation has 1.6-2.7X worse energy consumption than fixed-point. This is because the larger area and latency of parallel counter addition cannot compensate for the increased latency. The proposed OR_2 and OR_3 designs successfully bridge the performance gap between OR_1 and binary accumulation. OR_2 achieves up to 4.2X energy and 2X latency improvement over fixed-point at 32-long streams, and up to 2.4X at 64-long streams, with similar latency. OR_3 fares marginally worse, with up to 3.8X energy and 1.9X latency improvement at 32-long streams, and 2.2X improvement at 64-long streams. Both designs outperform SC with binary accumulation, even at twice the stream length. OR_2 and OR_3 with 64 long streams have up to 3.6X and 3.2X lower energy, respectively, than the parallel counter implementation with 32 long streams. They also outperform uSADD and uNSADD configurations at the same stream lengths, although it comes mainly from the high cost of accurate uMUL multipliers. With an AND-based multiplication, uSADD and uNSADD would be very similar to regular binary accumulation, as they are based on parallel counters.

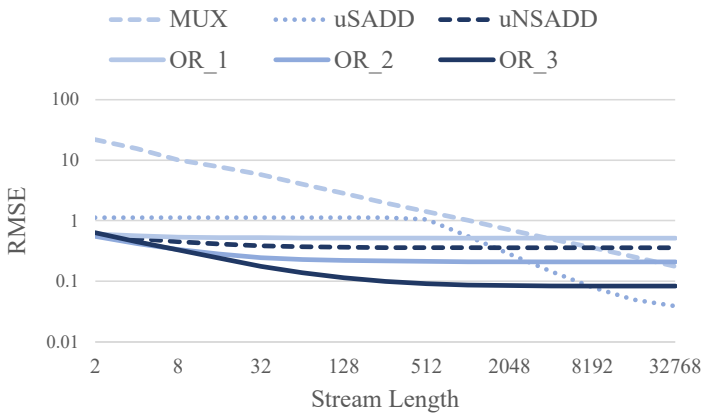



Figure 5.15: Comparison of multiplexer- and OR_n-based adders' root mean squared error (RMSE) with respect to normal addition. We use sums of 1000 positive inputs with outputs having an average of 1 and a variance of 0.5.

### 5.3.5 OR_n for non-trained applications

Most of the results focus on deep learning performance of OR_n with training involved. This is valid for neural networks that rely on training, and where the non-linearity of OR_n accumulation can be accounted for. In applications where training is not allowed, OR_1 already offers lower error compared to Multiplexers for relatively short streams as shown in Fig. 3.2. OR_2 and OR_3 extend this lead to even larger values, as shown in Fig. 5.15. MUX requires stream length $\geq 4096$ to have a lower error than OR_1 before factoring training. For OR_2 and OR_3, that threshold becomes 32768 and 131072 respectively.

We also compared the error of OR_n accumulation with unary computing accumulators proposed in [WLY20]. Despite not having the non-linearity of OR_n, the scaled addition version (uSADD) only beats OR_2 and OR_3 with stream length $\geq 4096$ and 8192 respectively. On the other hand, the non-scaled version (uNSADD) quickly saturates and is comparable to OR_1. Both versions are more expensive than SC_Bin discussed previously. At stream lengths used for accuracy and performance evaluations (32 and 64), OR_2 and OR_3 offer 1.57x to 8.10x lower average error.

## 5.4 Conclusion

In this work, we presented REX-SC - Range-Extended Stochastic Computing for neural network acceleration. REX-SC improves the accuracy of SC accumulation by increasing the number of output bits after accumulation. It reduces the accuracy gap between SC using OR accumulation and fixed-point computation. While SC with binary accumulation has almost no performance advantage over fixed-point, REX-SC remains competitive in performance with fixed-point neural networks while preserving the benefits of stochastic computing, including variable precision [RLM20] and early termination[WLY20]. REX-SC also improves the training performance of SC-based neural networks with optimizations to both the forward and backward propagation components.

# CHAPTER 6

# Solving the Training Issue[1]

The previous chapters on stochastic computing briefly touched on the issues surrounding training but didn't go into the specific details and challenges faced when training a neural network for execution on SC hardware. This chapter will provide a basic introduction to training for stochastic computing and approximate computing in general, followed by optimizations to improve the different components in the training pipeline.

Most neural networks are trained using variants of gradient descent. In the forward pass, each layer performs convolution, linear, pooling, or other operations present in the layer, and stores the output for use in the backward pass. In the backward pass, each layer calculates gradients with respect to the activations and weights of that layer. Weight gradients are used to update the weights, and activation gradients are used to calculate gradients for the previous layer. Training for stochastic computing is based on a variant of this approach called a straight-through estimator (STE). STE is originally designed for training quantized neural networks, especially binarized ones. Compared to training for floating-point computation, the forward pass is replaced with an accurate model of the quantized computation. The backward pass is not modified and still uses floating-point computation. As quantization functions are not differentiable by default, it is not practical to backpropagate through the quantizers. Instead, the quantizer is assumed to be a bias-free estimator of the floating-point

[1]**This work is performed in collaboration with Shurui Li and Puneet Gupta. This chapter contains material previously used in [LLG23]. My contributions to this work include the accelerated simulation for SC, the activation proxy modeling, and the error injection training implementation for stochastic computing and analog multiplication.**

computation and ignored in the backward pass.

The change to the forward pass can be easily adapted to SC. Instead of an accurate model of quantized computation, SC training performs an accurate model of SC computation in the forward pass. As the accurate modeling happens for **every** forward pass, simulation efficiency directly affects overall training time. Performing fast and accurate simulation of SC in the forward turns out to be a complicated software optimization problem and Sec. 6.1 will describe attempts at improving SC simulation efficiency.

Even after extensive optimization, a software simulation of SC proves to be the major training bottleneck, so there still needs to be reduced reliance on simulation speed. The backward pass of SC also requires special attention, as the backward pass of STE cannot be directly adapted to SC. Compared to normal quantization, accurate floating-point computation is **not** a bias-free estimator of stochastic computing. Especially for SC accumulation using variants of OR gates (OR_1, OR_2, OR_3, partial binary, etc), there is a non-negligible bias from the accumulation. Sec. 6.2 will discuss how a point-wise activation function can be used as a proxy for non-linear computation, and how the activation function is augmented with error injection to reduce accurate software simulation in the forward pass.

Apart from the simulation improvements, other improvements are also applicable to other approximate computing methods as well. We focus on three types of approximate computing methods to showcase the benefits of our approaches:

- Stochastic computing. We use linear feedback shift registers for stream generation, AND gate for multiplication, and OR for addition, which is similar to the setup in [RLM20]. We use 32-bit split-unipolar streams (64 total bits).

- Approximate multiplier. We use an approximate multiplier from EvoApproxLib [MHV17b]. Specifically, we use the mul7u_09Y setup. It is a 7-bit unsigned multiplier in the Pareto optimal set for mean-relative error, which is suitable for neural networks and has 8-bit input precision when combining the sign bit. Compared to an accurate 7-bit multiplier,

the approximate multiplier has 4X lower error and power consumption.

- Analog computing. To make a more generic argument, we set the effective array size of analog accelerators such that the partial sum of every convolution channel is quantized by the ADC. That is 9 for Resnet-tiny (Resnet-18 shrunk for TinyML applications used in [BRT21]) and Resnet-18 [HZR16], and 25 for TinyConv (a four-layer CNN used in [LSC18]). Since analog accelerators usually only support positive inputs and weights, we use split-unipolar accumulation in our training setup to handle negative weights, which results in 2× computation. We assume 4-bit ADCs are used for all evaluations in this paper. A relatively low ADC bitwidth is selected since the ADCs are usually the power bottlenecks of analog accelerators, therefore low-bitwidth ADCs are always preferred if the accuracy impact is not significant. The bitwidth for inputs and weights is set to 8-bit for all cases to focus on the impact of partial sum quantization.

While there are other approximate hardware implementations, we limit our study to the above three methods as they cover a wide variety of approximate computing. Our method applies to other approximate hardware setups with minimal change.

## 6.1 Speeding up SC simulation

The biggest bottleneck in SC training involves efficient simulation of stochastic computing computation in hardware. SC computation is very different from floating-point and fixed-point computation. The first issue is that randomness in SC introduces error in the computation. Despite fixing the generation pattern in the generation optimization steps in Chap. 4, SC computation is not completely accurate, and outputs can vary depending on the seed used to generate the inputs. For instance, for a multiplication that generates 0.06 for floating-point computation, the SC computation results depend on the input values and seed values. On the one hand, the seed values are different for different input positions, and different seed combinations produce different SC multiplication and addition results. On

the other hand, the multiplication results are also dependent on the input value combination. 0.3 times 0.2 is different from 0.1 times 0.6 in SC, despite both generating close to 0.6 for floating-point computation. These complications mean that it is expensive to model SC computation accurately. Despite the complications in modeling SC computation, such modeling is necessary. As will be shown later, a model trained from floating-point or fixed-point computation cannot be directly used for stochastic computing. Failing to model the SC computation correctly means an unusable model after training. Given the importance of SC modeling, custom kernels are developed to speed up SC simulation in the forward pass. While there are implementations for both x86 CPUs and Nvidia GPUs, most optimization work is performed for Nvidia GPUs due to their higher performance in other parts of the training pipeline. The following sections will describe some of the implementation and optimization details.

### 6.1.1 Stream packing

The first part of SC simulation implementation is determining how activation and weight streams should be packed. Since SC computation behaves differently compared to fixed-point or floating-point computation, an SC bit stream cannot be treated as a fixed-point number. I.e.: treating a 32-bit SC bit stream as a 32-bit integer and performing integer multiplications on it generates incorrect results. The individual bits in an SC bit steam do work similarly to fixed-point computation. Take an 8-way (8 unipolar input streams, 4 AND multiplications, 3 OR addition) multiply-accumulate (MAC) of 8-bit bit streams as an example. Each output bit can be viewed as the MAC results of the 4 corresponding input bits followed by a point-wise function. Fig. 6.1 demonstrates this concept. Performing simulation using such a method is functionally correct but highly inefficient. Most commercial CPUs and GPUs are optimized for 32-bit and 64-bit computation, and performing computation on only a single bit without vectorization significantly underutilizes the hardware. To have efficient SC simulation, the bit streams need to be packed to the native operator bit width

of the target hardware.

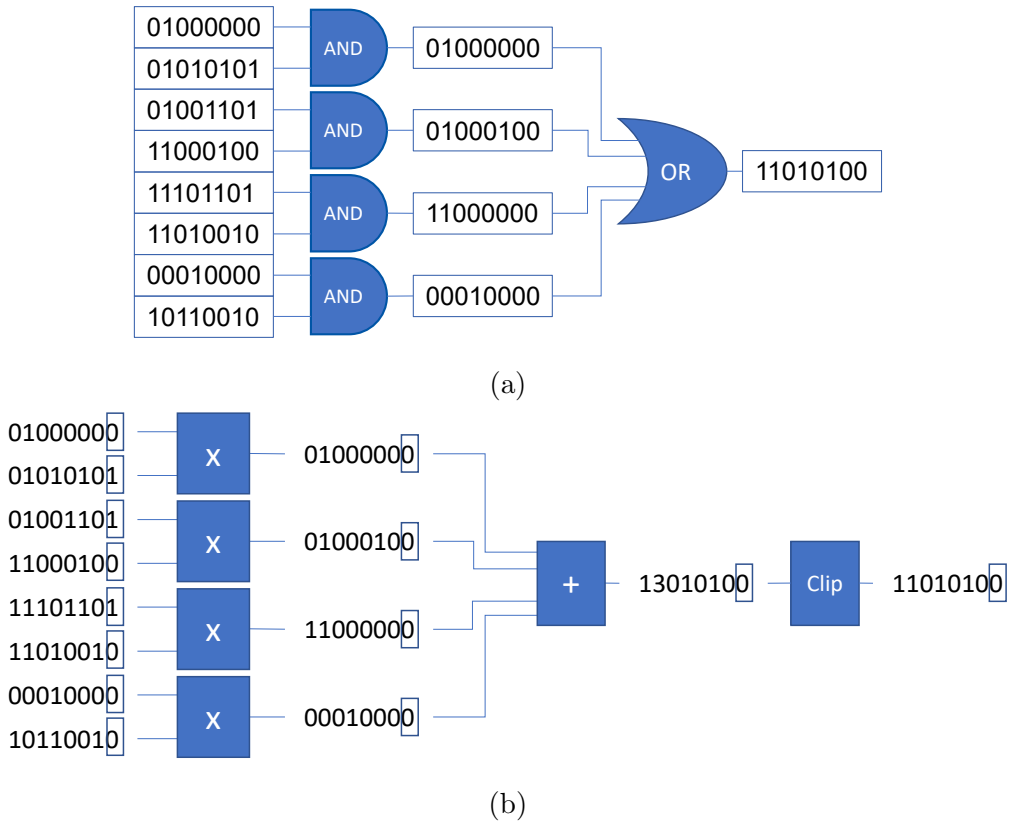



Figure 6.1: Comparison between (a) SC AND-OR dot product and (b) an equivalent implementation using single-bit multiply-add operations.

Since the priority target hardware is GPU for training SC-based neural networks and Nvidia GPUs are optimized for 32-bit computations, we choose 32-bit integers as the packing size. To efficiently utilize the packed values, packing needs to follow several constraints, as listed below:

- The packed dimension should allow SIMD-style computation. In other words, an integer SC operation (AND multiplication, OR_n or partial binary accumulation) should work on all 32 elements in the integer container.
- The packed dimension should be consistent between activation and weight tensors.

This is an extension of the previous constraint, as inconsistent packing activation and weight tensor makes SIMD computation on the packed values impossible.

Table 6.1: Operator and data format comparison between bitstream and input channel packing

| Property | Bit stream packing | Input channel packing |
|---|---|---|
| Unpacked shape | 64 (bit stream), 128, 256, 16, 16 (nchw) | |
| Packed shape | 128, 256, 16, 16, 2 (bit stream) | 64, 128, 16, 16, 8 (input channel) |
| Multiplication | bit-wise AND | |
| Addition | bit-wise OR | population count + sign |

With these constraints in mind, there are only two dimensions suitable for packing, the bit stream dimension and the input channel dimension (or k dimension for matrix multiplications of size (m,n,k)). Tab. 6.1 compares the properties of the two packing choices for activation values of a convolutional neural network, and Fig. 6.1a and 6.1b illustrates bit stream and input channel packing in boxes. While both satisfy the basic constraints for packing, packing in the bit stream dimension has some obvious benefits. Stream generation is much simpler with bit stream packing. Fig. 6.2 compares the generation scheme for bit stream and input channel packing. Whereas each thread for bitstream packing takes a single binary value and a single seed value and generates #(bitstream / 32) output integers, input channel packing requires 32 binary values, 32 seed values, and generates #bitstream output integers. Since outputs are stored in 32-bit containers (int), all 32 binary values associated with the 32 positions need to be tracked during generation. Since stream generation requires tracking the change in seed values, the corresponding seed values also need to be tracked. Tracking so many values on a single thread limits occupancy on GPUs, whether these values are treated as registers or stored inside local memory. Another factor favoring bit stream packing is the operation cost of additions. Bit-wise OR is a simple logical operation and operates at

full rate on Nvidia GPUs. On the other hand, population count is a special operation that operates at quarter rate (16 ops/clk/sm vs 64 ops/clk/sm). The last factor is that the input channel dimension is not always efficient to pack. The first layer of convolutional neural networks typically has only 1 or 3 input channels. Input channel packing would require padding the input channel to 32, which greatly reduces computation efficiency. Because of these factors, the initial implementation used for GEO in Chap. 4 is performed using bit stream packing and can be found in the repo here `https://github.com/nanocad-lab/geo`.

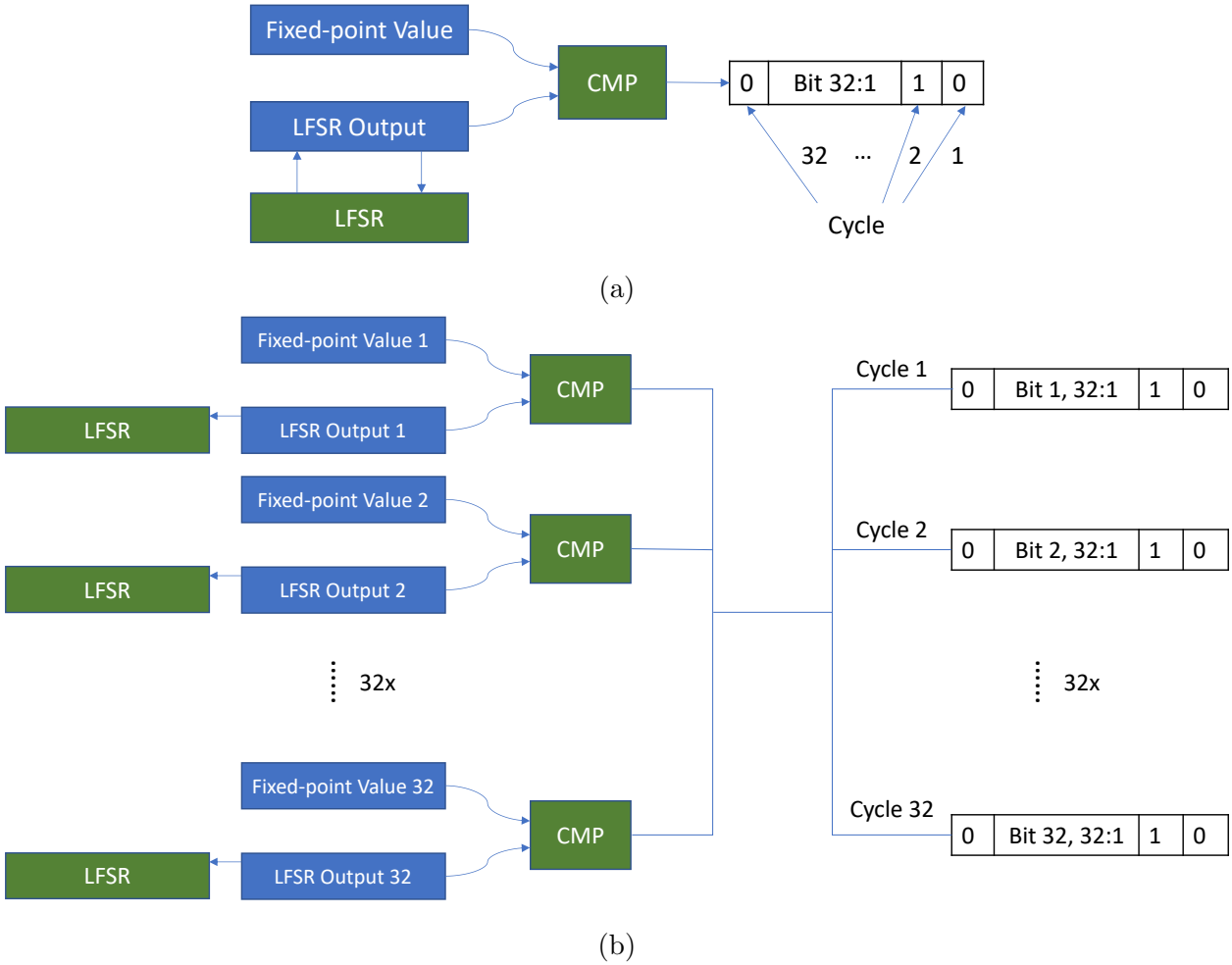



Figure 6.2: Comparison between stream generation for packing in (a) bitstream and (b) input channel dimension for a single thread.

Despite the benefits of bit stream packing and the initial implementation success, several factors changed the packing dimension to the input channel for later implementations. The first is the release of Nvidia's Ampere family of GPUs. Ampere GPUs introduce single-bit Tensor Core operations where multiplications are performed using AND operators. Previous versions of Tensor Core in Volta and Turing architectures already support single-bit tensor core operations, but the multiplication operator is limited to XOR operators typically used in binarized neural networks. As is mentioned in Chap. 3, split unipolar representation is needed for good accuracy using stochastic computing and allows the use of accumulation using OR gates. This representation rules out XOR gates as an SC multiplier. The Tensor core implementation in the Ampere architecture changes this, and the improved throughput from Tensor cores is tempting for SC simulation. The other factor is the introduction of OR_n accumulation in Chap. 5. While normal OR accumulation using OR gates for accumulation maintains output stream length and partial binary accumulation maintains stream length until the binary accumulation point, OR_n expands stream length to n times the input stream length for every n inputs and then maintains the stream length. This behavior undermines the benefits of cheap OR operators offered by OR and partial binary accumulation since logic OR is no longer sufficient for the accumulation operation. Instead, it favors the population count + sign accumulation pattern of input channel packing. For different OR_n values, only the sign function needs to be changed to accommodate increasing output range. As a result, the final implementations targetting OR_n accumulation use input channel packing instead.

With the packing structure determined, the overall SC simulation kernel is divided into four functions: activation stream generation, weight stream generation, stream computation, and output conversion. Activation and weight stream generation share most of the concepts and optimization and will be discussed together. To save memory write costs, output conversions are fused with stream computation and will be discussed together as well. Since the implementation assumes maximal stream reuse, stream computation is the main bottleneck most of the time.

### 6.1.2 Computation optimization

Since one of the main driving factors of using input channel packing is utilizing the new Tensor Core instructions on the Ampere architecture, most of the computation optimization centers around better utilization of the tensor cores. That said, the code still needs to work on older architectures, and the two versions (tensor and non-tensor) are written to share as much of the code as possible. The overall computation kernel can be roughly broken up into three components: activation and weight reshape, matrix multiplication, and output storage.

#### 6.1.2.1 Matrix multiplication

To simplify development and enable more reuse between different use cases, most of the optimizations are performed on a general matrix multiplication (GEMM) kernel for stochastic computing operations. A GEMM kernel can be directly used for linear layers, and convolutional layers can also be mapped to GEMM kernels, thus saving the time to optimize separately for convolution and linear. While the final implementation is a result of multiple iterations going from top to bottom and from bottom to top, the following paragraphs will describe the design choices in a bottom-to-top sequence for clarity.

The smallest unit for matrix multiplication on Nvidia GPUs is a Tensor Core operation. Directly programming the Tensor Cores requires using the CUDA C++ extension or PTX assembly, with the C++ extension easier to use. While simple, the C++ extension has several limitations:

- Limited data loading flexibility. Loading data into matrix fragments is handled using the load_matrix_sync function, which expects that the different rows/columns of the matrix are spaced by the same stride value. On the PTX side, data loading can be handled using the "ldmatrix" instruction, which allows arbitrary addresses for the starting point of each row/column. The limitations of the C++ extension play a row

in avoiding bank conflicts, which will be covered later.

- Limited matrix size support. Stochastic computing usinOR_n accumulation uses the binary tensor cores introduced in the Ampere generation but is only partially supported in the C++ extension. The PTX mma instruction supports 3 shapes of binary matrix multiplication, including (8,8,128), (16,8,128), and (16,8,256). In contrast, the C++ extension only supports (8,8,128). Lower compute density per instruction of the (8,8,128) version increases the number of instructions needed for data loading and computation and increases pressure on the instruction queue.

While the lack of matrix size support has a relatively small effect on performance, the inflexible data loading affects bank conflicts when using shared memory, which will be discussed later. As a result, the lowest-level kernel use PTX assembly to program the Tensor Cores directly. Using PTX also allows the densest (16,8,256) instruction for matrix multiplication.

Each Tensor Core operation requires the participation of an entire warp (32 threads). Single matrix multiplication requires two input matrices and an output matrix, all of which are stored in registers. For the (16,8,256) version, the two inputs and output matrices occupy 4, 2, and 8 32-bit registers respectively, so a single matrix multiplication requires 6 register reads. By loading two input matrices for each multiplicand, 4 matrix multiplications can be performed with 12 register reads, so the average register reads required per matrix multiplication can be reduced to 3 from 6.

The next level in the memory hierarchy above registers is shared memory, which acts like a fast scratchpad memory on Nvidia GPUs. Shared memory is arranged into 32 banks so that each thread in a warp can access different banks at the same time, which improves memory bandwidth. This also means that bank conflicts when different threads in a warp access the same memory bank (but not the same location), and the memory access needs to be broken up into multiple instructions, lowering effective bandwidth.

Fig. 6.3b shows the memory layout chosen for 8 warps in shared memory for (8,8,128)

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |
| 1 | 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |
| 2 | 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |
| 3 | 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |
| 4 | 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |
| 5 | 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |
| 6 | 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |
| 7 | 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |

(a)

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |
| 1 | 28 | 29 | 30 | 31 | 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 |
| 2 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 |
| 3 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
| 4 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 10 | 11 | 12 | 13 | 14 | 15 |
| 5 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 10 | 11 |
| 6 | 08 | 09 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 |
| 7 | 04 | 05 | 06 | 07 | 08 | 09 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 00 | 01 | 02 | 03 |

(b)

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 32 | 33 | 34 | 35 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 00 | 01 | 02 | 03 |
| 1 | 04 | 05 | 06 | 07 | 08 | 09 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 |
| 2 | 08 | 09 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 10 | 11 |
| 3 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 10 | 11 | 12 | 13 | 14 | 15 |
| 4 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
| 5 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 |
| 6 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 |
| 7 | 28 | 29 | 30 | 31 | 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |

(c)

Figure 6.3: (a) Naive memory layout, (b) strided memory layout used, and (c) padded memory layout for an 8x32(32-bit) array. The colored cells are for data storage, and the rest are either index (first row and column) or unfilled storage in (c). The numbers in the cell have slightly different meanings. In (a), the number represents both the column index in a column-major matrix and the bank index, as the number of columns matches the number of banks. In (b), the number represents the column index, and the bank index matches the column number. In (c), the number presents the bank index, and the column index matches the column number.

matrix multiplication shape. Different colors represent different matrix fragments, and each one is loaded using a "ldmatrix" instruction. The binary matrix multiplication instructions mandate row-major format for matrix a and column-major format for matrix b, which means that the inner dimension for both matrices is the one where dot products are performed, which also means that the same memory layout can be used for both matrices. The shared memory is shown as a matrix with 32 32-bit storage rows (row-major, matrix a) / columns (column-major, matrix b) as the inner dimension. Since there are 32 banks, and each bank is accessed at 32-bit granularity, the inner dimension index is the same as the bank index, and bank conflicts occur when one warp accesses the same column more than once. Fig. 6.3a shows the memory layout without modifying the data structure. As the same column is accessed 8 times (8 rows) for each warp, loading from shared memory incurs an 8-way bank conflict. In comparison, the strided layout in Fig. 6.3b avoids bank conflicts for all matrix loads from the shared memory, as different warps load data from different colored locations. Data is stored in shared memory in a row-wise fashion, where each warp loads one row of the shared memory. Since different columns correspond to different banks by design, shared memory stores also avoid bank conflicts. The 32x32 inner dimension of the shared memory corresponds to 128 bytes of data storage per row. Since Nvidia GPUs have 128-byte cache lines, this layout ensures coalesced data loading from L2 and global memory and efficient cache line usage [2].

The layout specified above requires 8 warps to load. This translates to 256 threads, which corresponds to the block size used for the GEMM kernel. The activation shared

[2]Later experiments with CUDA programming reveal that ensuring 128-byte coalesced load per warp is not always sufficient to ensure the most efficient use of L2 and DRAM. Data loading and storing can be done on a granularity of 4 32-bit elements/thread. In this setup, one warp loads 4 128-byte cache lines from a single data loading instruction, compared to 4 instructions when each thread only loads 1 32-bit element. Reducing the number of instructions needed prevents the instruction queue from being overwhelmed. From the experiments, using float4 or similar vector data types when loading global memory guarantees this more efficient instruction usage. Data access to shared memory has similar limitations. While ptx assemblies like ldmatrix and wmma.load ensures loading at a 128-bit granularity, loading single values from shared memory can trigger loading at lower granularities if data in shared memory is not structured correctly, and show up as "Stall MIO Throttle" when profiling.

memory allocated is (2*2*16, 1024b), which corresponds to 2*2*2 segments of the basic unit illustrated in Fig. 6.3b. The last 2 stands for the number of unique activation fragments used. During computation, the 8 available warps are organized as 4x2, corresponding to 4 weight fragments, 2 activation fragments, and 8 output fragments. The second 2 stands for the size of each activation fragment. Since we are using the (16,8,256) version of matrix multiplication instruction, each activation fragment has a shape of (16,256), which has twice the number of rows compared to (8,128) (the difference in the number of columns changes the number of iterations from 8=1024/128 to 4=1024/256). The first 2 stands for manual unrolling. During each iteration, 2 activation fragments are loaded from shared memory and multiplied with 2 weight fragments, accumulating to 4 output fragments. The weight shared memory allocated is (2*4*8, 1024) for both positive and negative weights from the split-unipolar setup, which corresponds to 2*4 segments of the basic unit. The 4 stands for the number of unique weight fragments used, and the 2 stands for manual unrolling.

The overall computation uses an output-stationary dataflow, as the outputs need to be updated to simulate the effect of OR_n once a single bit finishes processing. As the PTX version of WMMA instructions defines the output matrix fragments as registers, the OR_n update can be performed directly on the matrix fragments, and storing the matrix fragments back to global memory can also be done from the fragments directly without invoking the stmatrix instructions. Doing so also avoids allocating shared memory for output data.

The OR_n update step involves comparing the accumulator result from the output fragment with $n$, incrementing the output stream value by $\min(v, n)$ where v is the accumulation result before OR_n activation, and zeroing the accumulator. A naive implementation would require 2 32-bit registers, one representing the accumulator result and one representing the output stream. However, 16-bit is likely sufficient for the datasets used in this thesis. A 16-bit integer can represent a maximal value of 65535, which requires a dot product size larger than this value to have a possibility of overflowing. From this observation, a single 32-bit integer register is used for both the accumulator and output stream. The lower 16

bits store the accumulator results and act as the output matrix fragment. The upper 16 bits store the output stream and are updated using the OR_n update function.

The weight and activation shared memory occupy 24KB in total as described in the previous paragraphs. To overlap computation with global memory loading, two instances of shared memory are required, where one is used for computation and the other one loads data to be used in the next iteration. This brings the total shared memory per block to 48KB, which is right at the edge of static shared memory allocation. Combined with 1KB of shared memory allocated by the GPU driver, each block requires 49KB of shared memory and allows 2 blocks/sm on SM8.6 devices with 100KB of shared memory. This strict memory requirement is another reason for choosing the PTX version of ldmatrix, as neither the C++ nor PTX version of wmma.load allows different stride values between rows. Shared memory bank conflicts can also be avoided by padding each row to 36 32-bit elements from 32, as is shown in Fig. 6.3c. Since the padding method maintains the same stride value between rows, it can be implemented using wmma.load. Compared to the other two cases where the column number matches the bank index, padding each row by $n$ offsets the bank index by $n$ between rows, so a padding of 4 32-bit elements per row completely avoids bank conflicts. However, padding increases the shared memory required per block to 55KB, which reduces GPU occupancy by half. Since each block consumes 256 threads, 2 blocks/sm equates to 512 threads/sm, which is equivalent to 33% occupancy. The biggest culprit for low occupancy is 2x unrolling of the activation and weight components, which demand more registers to hold the input and output matrix fragments. Reducing the amount of manual unrolling reduces register pressure and improves occupancy, but leads to reduced performance due to less register reuse. Conversely, further increasing register reuse through more unrolling lowers occupancy further and also reduces performance.

Since shared memory competes with the L1 cache for capacity and the GEMM kernel carves out the largest size shared memory allowed, all of the data loading needs to come from the global L2 cache or DRAM. To avoid being bottlenecked by a slow DRAM interface,

most of the memory accesses should come from L2 instead of the global GDDR6(X) memory. Within the output window, each thread block processes a small 2D portion of it, and each 2D portion corresponds to one activation and one weight matrix. The row height of the activation matrix and column height of the weight matrix correspond to the dot product size and can be as large as $streamlength \times filtersize$. Keeping too large of a dot product can mean evicting earlier computed portions of the activation and weight matrices, which affects the L2 hit rate of neighboring thread blocks. To avoid this issue, the dot product size is limited to 512x32-bit for each GEMM kernel, which is denoted as K_UNIT_BLOCK. Since all compute-intensive layers are mapped to the GEMM kernel, both activations and weights are reshaped into matrices with their dot product size, which is denoted as k_compute. To ensure that the kernel has acceptable utilization for different layer shapes, k_compute needs to be as close as possible to K_UNIT_BLOCK as possible while being smaller. A large k_compute ensures that the number of output memory accesses is limited. The following lists a few rules used to determine k_compute:

- If filter size is not a multiple of 8x32, pad filters to a multiple of 8x32-bit equal to filter size padded. This step is used to ensure that (16,8,256) wmma instructions can be used correctly.
- If `filter size padded` $\times$ `stream length` $<= K\_UNIT\_BLOCK$, set k_compute to `filter size padded` $\times$ `stream length`.
- If `filter size padded` $\times$ `stream length` $> K\_UNIT\_BLOCK$, fill as many `filter size padded` segments as possible without exceeding K_UNIT_BLOCK.
- if filter `size padded` $> K\_UNIT\_BLOCK$, give up. Allowing filter sizes larger than the dot product size means that partial filter results need to be stored between dot product iterations, increasing memory activities. Since 512x32 is a relatively large size already and larger sizes will likely cause convergence issues for SC, larger dot products are not provisioned.

The other L2 cache consideration is thread block scheduling. If thread blocks are scheduled using a simple raster scan pattern, the first few weight matrices can get evicted out of the cache if the 2D window is very large. Since the 2D output window corresponds to reshaped input and weight tensors, they need to be kept large to amortize the reshaping cost, and it is not possible to fit the transformed input and weight matrices all in L2. To alleviate this issue, the thread blocks are ordered using a simple thread swizzle pattern, similar to the one mentioned in [Bav].

By default, memory for input and weight streams and temporarily transformed input, weight, and output needs to be allocated for each invocation of the kernel and freed after kernel execution. To avoid repeated memory allocations, a global pointer is used for the temporary memory, which points to a memory pool equal to the largest needed amount for all layers at the start of a training run. Allocating larger temporary buffers improves performance for large problem sizes at the cost of reduced efficiency for smaller problem sizes. The final total buffer is chosen to be $\approx$ 800MB with the activation matrix sized much larger than the weight matrix. In mini-batch training workloads, the activation size of a layer is typically larger than the weight matrix, so a larger activation matrix makes sense for performance concerns.

#### 6.1.2.2 Input reshape

Since the underlying kernel performs a GEMM operation, input and weight streams need to be reshaped to use the GEMM kernel. For convolutional layers, this step involves an image-to-column (im2col) transformation. The step to avoid bank conflict requires a modified data layout in shared memory. It is possible to perform this data layout change using asynchronous memory copy at a finer granularity by loading the same-colored segment in a row shown in Fig. 6.3b. However, invoking the memcpy_async with a small group size significantly drops copy performance. Nvidia GPUs operate in SIMD32 mode, so a group size smaller than 32 underutilizes warps. The L2 cache has a cache line of 128 bytes, which equates to 32

32-bit elements. The compiler might not be combining multiple memcpy_async calls to a single cache line read. To avoid this cost, the data layout change is performed in the input reshape step. While both im2col and layout change constitute additional overheads, the input reshapes step only constitutes less than 15% of the overall runtime.

### 6.1.3 Generation optimization

With the improved performance of the computation kernel, the overall bottleneck shifts slightly to the stream generation components. As mentioned in Sec. 6.1.1, input channel packing requires a lot of registers/shared memory resources per thread, which reduces occupancy. Since we are targeting a low stream length of 32-64 for our SC implementation, the corresponding seed and binary value precision are also limited to 5 or 6 bits. Instead of 32-bit integers, the binary values and seed values are stored as 8-bit integers which are still plenty for our use case. This change improves occupancy of the activation stream generation kernel from 20% to 80%, and the weight stream generation kernel from 7% to 15%. While the weight stream generation kernel still has low utilization, it has a lower impact on the overall runtime. While neither kernel is particularly memory nor compute-bound after optimization, they are much cheaper than computation, and more optimizations will lead to negligible improvement to the overall training performance.

## 6.2 Training Neural Networks for Execution on Approximate Computing Hardware

### 6.2.1 Activation proxy function

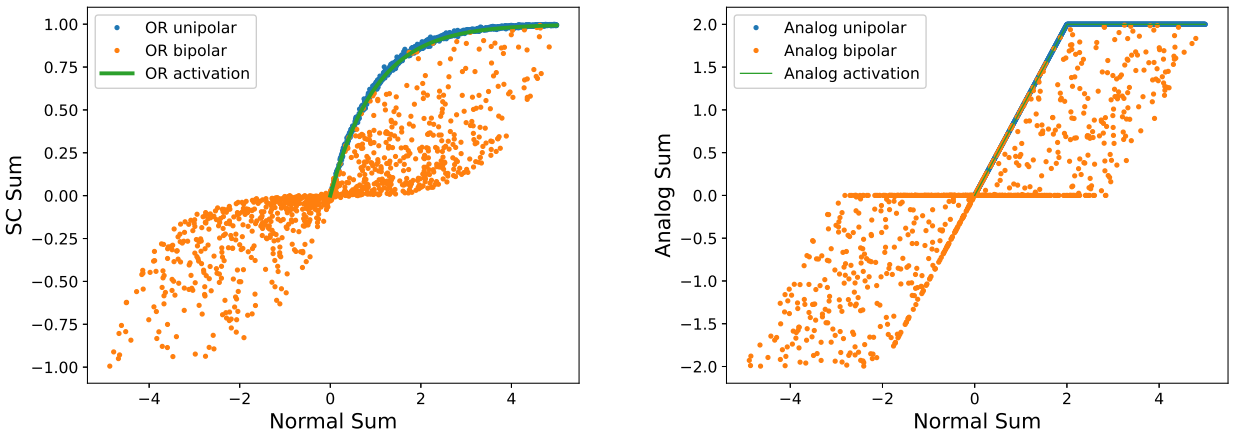
One issue of inaccurate computation is the non-linearity introduced to multiply-accumulate operations, which is separate from non-linear activation functions like ReLU. We use stochastic computing and analog computing as examples. Approximate multipliers do not suffer

from the non-linearity issue, as errors are only introduced during multiplication. For SC, if the two input values $a$, and $b$ are uncorrelated, the OR adder performs $a+b-ab$ on average. For analog computing, the partial sum and output results need to be clamped and quantized. Accurately modeling the imperfections in computation can be costly in the backward pass. For instance, stochastic computing using OR adder requires tracking almost all inputs in the adder ($\frac{\partial}{\partial a_i}\text{OR}(a_j) = \prod_{j\neq i}(1-a_j)$) during backpropagation, whereas accurate addition is much simpler in the backward pass ($\frac{\partial}{\partial a_i}\text{Sum}(a_j) = 1$). On top of being expensive to model, the nonlinearity can hinder convergence if it is not taken into consideration. Using activations as a proxy during backward pass is first proposed in[RLM20] to simulate the effect of SC OR accumulation. *Contrary to normal activation functions, the added activation proxy is only used during training in the backward pass and is not used during inference.* With proper implementation of the activation function, the overhead of the activation function can be negligible. As is shown in Tab. 6.2, modeling the non-linearity using an activation function is necessary. TinyConv is a four-layer CNN used in [LSC18], and Resnet-tiny is Resnet-18 shrunk for TinyML application used in [BRT21]. Models are trained with accurate modeling of stochastic and analog computing in the forward pass. For analog computing, accuracy is noticeably lower when trained without the activation function. For stochastic computing, training does not converge at all without the activation function.

Table 6.2: Accuracy benefits of using activation functions.

| Setup | TinyConv | Resnet-tiny |
|---|---|---|
| Stochastic Computing | | |
| No Activation | 41.64% | 10.00% |
| With Activation | 72.18% | 79.76% |
| Analog Computing (4-bit) | | |
| No Activation | 74.85% | 69.21% |
| With Activation | 79.20% | 77.23% |

Using an activation function requires the computation to be (mostly) associative. Take




(a) (b)

Figure 6.4: Activation modeling behavior of unipolar and bipolar (a) stochastic and (b) analog computation. The bipolar versions performs subtraction between two unipolar (positive and negative) inputs to achieve the full range. For analog computation, the ADC saturation is modeled as a clamp at 2 in this example, and other effects (e.g. size of accumulation) are not considered.

the previously used OR accumulation and analog computing as examples. Computation is broken up into positive and negative parts since both types of approximate computing work on unipolar (positive-only) inputs. Accumulation within each part is associative, but subtracting the two parts isn't associative with the rest of the accumulation. This behavior can be visualized in Fig. 6.4. While the activation function can approximate unipolar OR accumulation, a single activation cannot be used to model the entire accumulation when subtraction is factored in. As such, accumulations need to be split into positive and negative parts in the backward pass, so that each part can be modeled using accurate accumulation with an activation function. Similar effects can be seen in the analog computing setup. Since the positive and negative parts saturate individually, a single activation function is also insufficient. Tab. 6.3 lists the activation functions used for stochastic computing and analog computing.

Table 6.3: Activation functions for stochastic computing and analog computing. $x$ is the output of a layer before activation. $x_{pos}$ is the output of positive weights and $x_{neg}$ is the output of negative weights. Inputs are assumed to be non-negative due to ReLU activation.

| Method | Activation Function |
|---|---|
| Stochastic Computing | $\text{SC\_act}(x) = (1 - e^{-x_{pos}}) - (1 - e^{-x_{neg}})$ |
| Analog Computing | $\text{Analog\_act}(x) = \text{HardTanh}(x_{pos}) - \text{HardTanh}(x_{neg})$ |

### 6.2.2 Error injection training

Despite resolving the issue of backpropagation, the activation function method cannot fully replace forward propagation. Training models for non-floating-point computation typically involves modeling the computation accurately in the forward pass, be it low-precision fixed-point computation (including extreme precision like binarization [ROR16]) or approximate computation [GSG22]. For fixed-point computation, modeling the computation in the forward pass is relatively cheap. An element-wise fake-quantization operator followed by a normal convolution/linear operator is sufficient. The same cannot be said for approximate computing methods. SC requires emulating the stream generation and bit-wise multiplication in hardware. Approximate multipliers require hundreds of lines of bit manipulation. Analog computing requires emulating the limited hardware array size and ADC precision during computation. Tab. 6.4 demonstrates the high cost of emulating approximate computing. In all cases, emulating the computation is expensive, requires additional coding, and cannot be completely overcome even with abundant programming resources.

While it is expensive to model approximate computing accurately in the forward pass, skipping modeling degrades accuracy, as shown in Tab. 6.5. Despite being sufficient for the backward pass, the activation method defined in Sec. 6.2.1 is not sufficient for the forward pass. To combat this limitation, we propose to replace accurate modeling with error injection coupled with normal training.

Table 6.4: Relative multiplication and addition cost. FP32 multiplication and addition are used as the baseline. The number of operations in C++ is used as the cost value.

| Method | Multiplication | Addition |
|---|---|---|
| Floating point | 0.5(fused) | 0.5(fused) |
| Stochastic Computing (32-bit) | 64(unrolled)<br>2(packed) | 64(unrolled)<br>2(packed) |
| Approximate Multiplication | 86 | 1 |
| Analog Computing | 1 | 1(within channel)<br>9(between channel) |

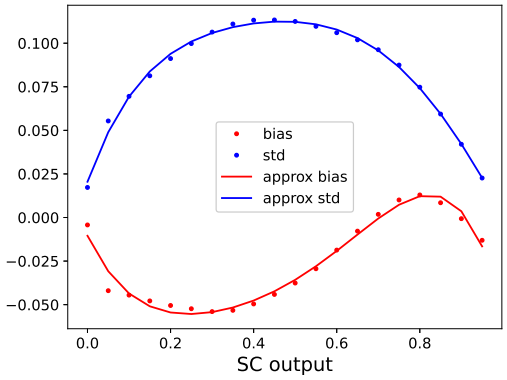
Table 6.5: Accuracy impact of modeling approximate computation. "With Model" models the approximate computation method accurately in the forward pass.

| | TinyConv | | Resnet-tiny | |
|---|---|---|---|---|
| Method | Inference Only | With Model | Inference Only | With Model |
| Stochastic Computing | 14.87% | 72.18% | 48.57% | 79.76% |
| Approximate Multiplication | 71.88% | 83.35% | 76.95% | 85.25% |

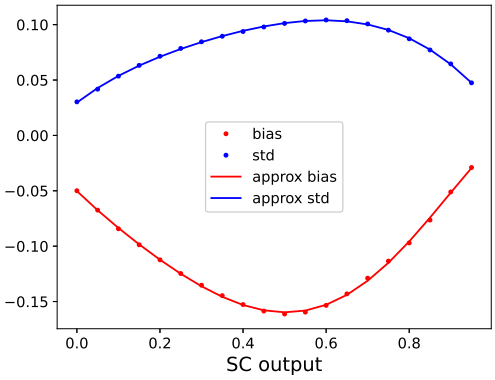
The error injection training models the difference between the activation proxy function and accurate modeling as functions of the output values of a layer, and we use it for stochastic computing and approximate multiplication. Fig. 6.5 shows the error average and variation of the four layers of TinyConv using stochastic computing. Compared to the value expected from the simple activation function $y = 1 - e^{-x}$ for OR accumulation, the actual layer output differs significantly. The variance (represented as "std") in the plots means that it is impossible to fully capture the computation details using only an activation function. Given that the activation function is only an approximation by design, it cannot capture all the characteristics of the inaccurate computation. The non-zero average error means that the activation function doesn't represent the target computation accurately on average. Activation functions are derived under certain assumptions. In the case of OR accumulation, the assumption is that the size of accumulation $n$ is large and all input values are small and similar in value. These assumptions don't always hold in a real model, and the difference in average error between layers means it's impossible to have a single activation function for the entire model. For deep models, the error of the activation function accumulates and results in non-usable models after training. To resolve this issue, we propose to add correction terms to the activation function during training.

Take the error profile shown in Fig. 6.5 as an example. The mean and variance of the error are plotted with respect to the output after the activation function, and both appear to be smooth functions. From this observation, we model the average as a function of activated output values on top of the original activation function, and the variance as a normally distributed random error with variance dependent on the activated value. Both curves are fitted to a polynomial function that's different for each layer and calibrated 5 times per epoch. Though it is possible to lower it further, this frequency sufficiently amortizes the cost of error calibration.

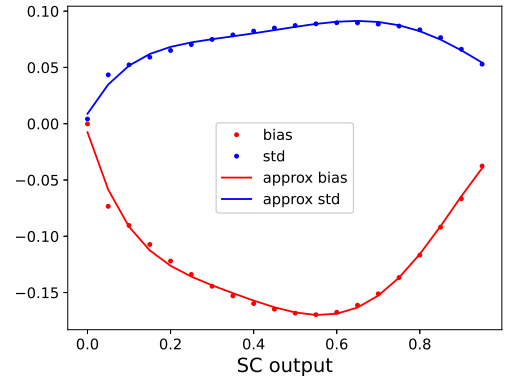
Using error injection in the place of accurate modeling improves accuracy compared to not modeling at all, as shown in Tab. 6.6. However, training with error injection alone



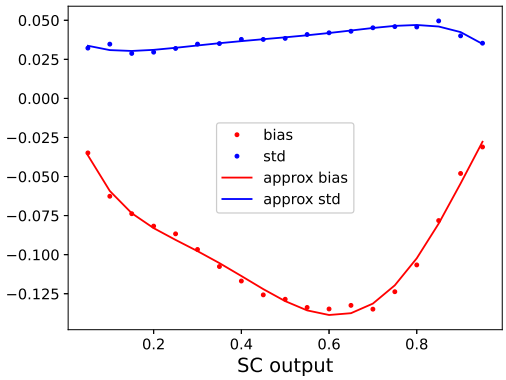

(a) Layer 1




(b) Layer 2




(c) Layer 3




(d) Layer 4

Figure 6.5: Difference between outputs from stream computation and normal computation+activation.

is not sufficient to completely bridge the gap, especially for analog computing with a low ADC bitwidth. Later we will show the accuracy gap can be eliminated through a fine-tuning step. Runtime is improved up to 6.7X with error injection compared to using an accurate model during training, as shown in Tab. 6.8. In most cases, the error injection runtimes are comparable to those in normal training (the "Without Model" column).

Table 6.6: Accuracy impact of error injection training.

| | TinyConv | | | | Resnet-tiny | | | |
|---|---|---|---|---|---|---|---|---|
| Method | Inference Only | With Model | Error Injection | Fine-tuning | Inference Only | With Model | Error Injection | Fine-tuning |
| Stochastic Computing | 14.87% | 72.18% | 64.01% | 73.09% | 48.57% | 79.76% | 76.71% | 81.29% |
| Approximate Multiplication | 71.88% | 83.35% | 79.06% | 83.05% | 76.95% | 85.25% | 81.06% | 84.85% |

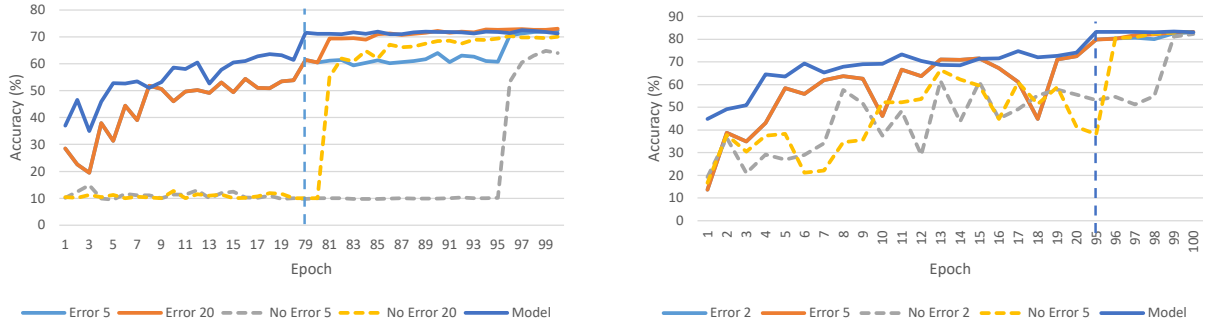
### 6.2.3 Fine tuning

While the error injection mentioned in Sec. 6.2.2 cannot achieve the same accuracy as training with accurate modeling, it reduces the amount of fine-tuning with accurate modeling required to achieve the same accuracy. The goal of error injection is to simulate the error during training, which can make the model more robust to errors. However, since the injected error is randomly generated while the actual error is input-dependent, error injection cannot achieve the same accuracy as accurate modeling. Still, error injection makes the model more robust such that with a small amount of accurate modeling, the model weights can quickly converge to the optimal point.

Fig. 6.6a compares the convergence behavior when error injection is combined with accurate modeling for SC when trained for CIFAR-10. For stochastic computing, error injection combined with 5 epochs of fine-tuning is sufficient to achieve the same accuracy as using an accurate model throughout the training. In contrast, convergence is poor without error injection, and the model does not converge to the same accuracy even with 20 epochs of fine-tuning.

Fig. 6.6b compares the convergence behavior for the approximate multiplier on the same

model. Since the accuracy gap is smaller with the approximate multiplier, training without error injection can also converge properly with 5 epochs of fine-tuning. However, error injection reduces that further to 2 epochs.




(a) (b)

Figure 6.6: Convergence behavior of TinyConv with and without error injection using (a) stochastic computing and (b) approximate multiplication. Stochastic computing and approximate multiplication models are trained from scratch for 100 epochs and the first 20 epochs and the last few epochs are shown. "Error X" means training with error injection with X epochs of fine-tuning with an accurate model. "No Error X" means training without error injection with X epochs of fine-tuning. "Model" means accurate modeling throughout training.

Overall, fine-tuning on top of error injection removes the accuracy deficit of error injection alone. As shown in Tab. 6.6, accuracy after fine-tuning ("Fine-tuning" columns) is at most 1% pt lower than training completely using an accurate model.

### 6.2.4 Gradient checkpointing

Both the activation function in Sec. 6.2.1 and error injection in Sec. 6.2.2 add additional steps in the computational graph during training, which increases memory requirement and limits batch size for larger models. To reduce memory consumption, we use the gradient

checkpointing setup from [CXZ16]. Since the added functions are point-wise functions and have low compute intensity (< 20 OPS/memory access), checkpointing all added computations has minimal effect on runtime even when not memory bound. Tab. 6.7 compares the runtime and memory consumption with and without checkpointing for Resnet-18 on the ImageNet dataset. Checkpointing allows training with a larger batch size, which improves GPU efficiency and reduces runtime required per epoch by 22% in this case.

Table 6.7: Training runtime and memory requirement of stochastic computing without and without gradient checkpointing. We use the maximum batch size achievable which is a power of 2.

| Setup | Memory (MB) | Batch Size | Runtime (s/epoch) |
|---|---|---|---|
| With Checkpoint | 19840 | 256 | 1326 |
| Without Checkpoint | 12766 | 128 | 1692 |

### 6.2.5 Results

In this section, we demonstrate the benefits of our techniques on a more complicated workload. We use Resnet-18 training on the ImageNet dataset to demonstrate the benefits. Models are trained on a single RTX 3090 using mixed precision. We use PyTorch 1.12 as the baseline framework and implement the additional operators as CUDA C++ extensions, including accurate modeling for stochastic computing and approximate multiplication. While we make the best effort to optimize the kernels used for modeling approximate computing, we cannot guarantee that the implementations are optimal. Models are trained from a checkpoint provided by PyTorch to reduce training time, which uses fp32. Tab. 6.9 lists the detailed training setup.

Tab. 6.10 shows the accuracy achieved for all three setups. While there are no previous accuracy results that we can compare against, the accuracy results follow the same trend as

Table 6.8: Runtime impact of error injection training. Shown is the time (seconds) required per epoch. Runtimes are measured on an RTX 3090 using TF32 precision and batch size=256. SC and approximate multiplication are slower due to the need to split positive and negative computations discussed in Sec. 6.2.1. Fine-tuning runtime is the same as the runtime with accurate model (the "With Model" column).

| Method | Without Model | With Model | Error Injection |
|---|---|---|---|
| | TinyConv | | |
| Stochastic Computing | 3.86 | 9.50 | 3.90 |
| Approximate Multiplication | 3.86 | 28.3 | 4.20 |
| Analog Computing (4b) | 2.13 | 3.91 | 2.73 |
| | Resnet-tiny | | |
| Stochastic Computing | 8.51 | 13.3 | 8.65 |
| Approximate Multiplication | 8.13 | 38.4 | 9.11 |
| Analog Computing (4b) | 4.88 | 8.51 | 6.53 |

Table 6.9: Epochs used for training. Methods with higher accuracy require fewer epochs of fine-tuning.

| Method | Error Injection | Fine-tuning |
|---|---|---|
| Stochastic Computing | 30 | 5 |
| Approximate Multiplication | 34 | 1 |
| Analog Computing (4b) | 14 | 1 |

that seen for smaller models discussed in previous sections. Approximate multiplication and analog computing take too long to train without our improvements, as we will show later.

Table 6.10: Top-1 accuracy of Resnet-18 on ImageNet.

| Method | Without Improvements | With Improvements |
|---|---|---|
| Stochastic Computing | N/A | 54.29% |
| Approximate Multiplication | N/A | 67.81% |
| Analog Computing (4b) | N/A | 63.40% |

Fig. 6.7 showcases the performance benefits of the proposed methods when combined. The improvements shown here underestimate the benefits of the proposed methods as models are validated after each epoch. Since validation uses accurate modeling and is the same with and without improvements, the performance gap will be even more significant if the validation frequency is reduced. Despite this limitation, our methods reduce end-to-end training time by 4.6X to 250X. In all three cases, the benefits are even larger than for the smaller models on CIFAR-10. Approximate multiplication especially benefits from error injection, as the iteration time reduces by 36.6X compared to accurate modeling. For stochastic computing, the runtime result without improvements also removes the activation proxy function. Without the activation proxy, all additions need to be broken up during backpropagation for stochastic computing. For analog computing, the runtime difference on ImageNet is larger than that on CIFAR-10, as both TinyConv and Resnet-tiny cannot fully utilize the GPU on CIFAR-10.

## 6.3 Conclusion

This chapter discusses the efforts to improve the training speed of stochastic computing and other approximate computing methods. Though not extensively measured, the packed SC simulation kernel is 16X faster than the naive implementation that computes each

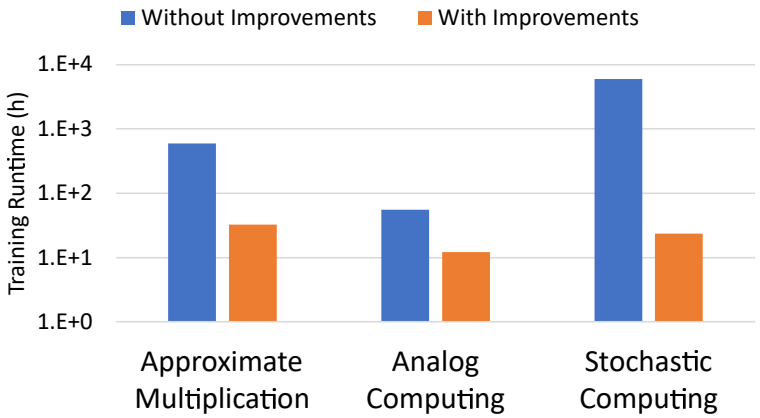



Figure 6.7: End-to-end runtime improvements. Time is measured in hours required to converge. The "Without Improvements" results are estimated for models that are impossible to train without the improvements.

bit separately. Through activation modeling, error injection with fine-tuning, and gradient checkpointing, we achieve convergence on a wide range of approximate hardware. These improvements combined make it feasible to train large models like Resnet-18 on the ImageNet dataset using a single consumer GPU.

# CHAPTER 7

# Learned Approximate Computing[1]

Neural network training can be viewed as training an application for a specific task. The computation performed in a neural network does not change before and after training in general, while the capabilities of the model improve a lot from the training process. Approximate computing methods including stochastic computing similarly benefits from training, as the training process can learn the error patterns of approximate computing and improve the final output quality. This chapter will discuss our efforts on improving other applications through the training process.

## 7.1 Introduction

Approximate computing trades some computation accuracy to improve area and energy efficiency. Some examples of approximate computing include introducing uncertainty by lowering voltage [HS99] or introducing input-dependent error to simplify logic design [KGE11, KWK10]. Despite finding interest in error-tolerant applications like deep learning using neural networks [VAR11], the inherent uncertainty in approximate computing has prevented wide adoption in mainstream applications.

Multiple approaches have been proposed to improve the accuracy of approximate computing. The methods include compensating errors with additional circuitry [MHT19, MHT20],

[1]**This work is performed in collaboration with Egor Glukhov, Vaibhav Gupta, and Puneet Gupta. This chapter contains material previously used in [GLG22]. My contributions to this work include the training methodologies for fixed-hardware and trained-hardware LAC.**

finding better accuracy-performance tradeoffs through better logic design [FAF13] or allowing multiple accuracy levels [KWK10, MHV17a, VS14]. As approximate computing relies on uncertainties to achieve better performance, reducing the overall error runs into diminishing returns.

To further enhance the accuracy of approximate computing in practice, there are two aspects to improve: the approximate computing hardware and the target application. Most recent works focus on optimizing approximate computing hardware for specific applications. The approaches include reducing the approximation error for computations prevalent in an application [BTF17, SB20, JGK15], or limiting approximation to specific error-tolerant components of applications [MCA14]. Optimizing the approximate computing hardware for one or two particular applications means that the hardware may not be suitable for other applications. Efforts on optimizing the latter have been sparse [GLG22], and most past works have focused on neural networks [VRR14, ZWT15]. With all the different approximate computing options, it also becomes challenging to choose the setup that is most suitable for a given application or a specific performance or quality constraint.

In this work, we propose LAC: learned approximate computing. Instead of optimizing the approximate computing hardware for a fixed application, we optimize the *application* kernel for both a fixed and trained hardware configuration. The key observation is that the application and the approximate computing hardware can be co-optimized to achieve optimal performance while minimizing manual intervention. During the process, we also allow searching for the approximate hardware configuration that is most suitable for the application. This way, the coefficients of an application kernel are automatically adjusted to the error properties of the most appropriate approximate hardware configuration. Our contributions are as follows:

- We develop the LAC methodology to train almost arbitrary parameterizable application kernels.

- In cases where the approximate hardware is fixed, LAC tunes the algorithm to better match the hardware.
- We show that LAC improves structural similarity (SSIM) by up to 0.96 and peak signal-to-noise ratio (PSNR) by up to 9.85dB on the same approximate hardware. The improved quality reduces power and latency by 87% and 83% for the same application quality.
- In cases where the hardware can be changed, LAC searches for the optimal hardware while tuning the algorithm.
- We show that LAC finds Pareto optimal solutions for various applications and performance constraints.

## 7.2 Fixed Hardware LAC

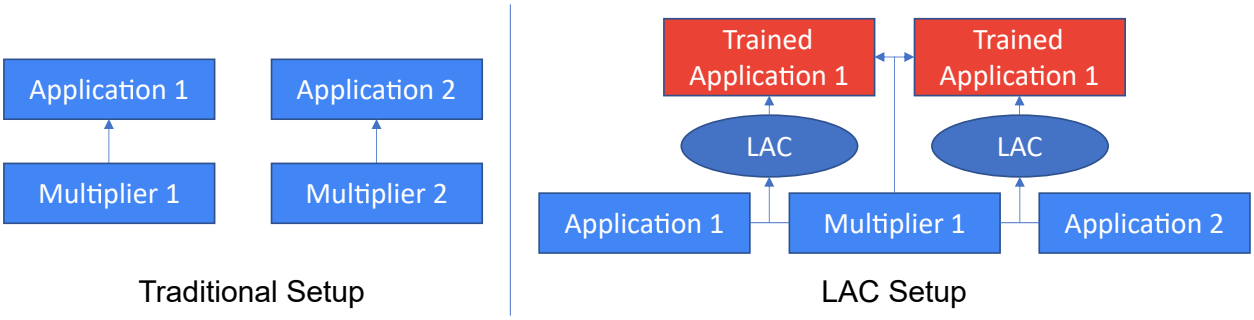



Figure 7.1: Difference between traditional and LAC setups. LAC focuses on optimizing the application kernels rather than optimizing the hardware approximations.

While most previous works try to reduce the error of the approximate hardware for a particular application, the application itself is mostly untouched. This approach means that the approximate hardware is not optimal for other applications, and different applications require separate approximate hardware for the best quality and performance. This section will focus on a fixed-hardware version of LAC where applications are trained for fixed hardware.

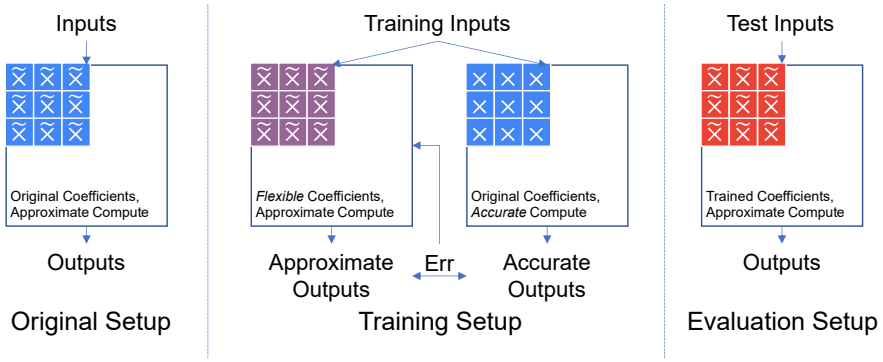



Figure 7.2: Overview of LAC for fixed hardware.

### 7.2.1 Training applications for a fixed hardware

In a fixed hardware setup, LAC tries to make the application learn the approximate computing hardware by training its parameters. Fig. 7.1 compares a traditional approximate computing setup with LAC. By training the application to better match the properties of the approximate multiplier, LAC aims to improve application performance for a specific multiplier and allow the reuse of approximate computing hardware across multiple applications. The motivation behind LAC is that error distributions in approximate computing units are strongly input-dependent. An example is the Kurkarni [KGE11] multipliers, which only have errors when multiplying three by three within a 2-bit multiplier and no error for any other multiplication. If an application is dominated by multiplying three by three, the results have very high errors. Conversely, if the application does not contain three in any 2-bit sections, multiplications can be completely accurate. This discrepancy means that approximate computing hardware does not have consistent performance across multiple applications since the values used between applications vary greatly. One way to get around this issue is by modifying the application coefficients so that the computation performed avoids the high-error regions of the approximate hardware. However, approximate compute units differ in their error characteristics. Dynamic Range Unbiased Multiplier (DRUM) [HBR15] lowers average error at the cost of introducing error in more multiplications. Even if it is possible to achieve better overall performance with a lower average error, manually tuning the coefficients of an

application to achieve that becomes difficult. LAC tries to simplify this process by training the coefficients. If a learning algorithm has access to the expected accurate result and all the properties of the approximate hardware, it should be possible to avoid the high-error regions. LAC is *not* limited to machine learning-type applications. The only constraint is that the application kernels should be parameterizable and hence be able to optimize using standard mathematical optimization approaches.

Figure 7.2 demonstrates the overall setup of LAC. Before training, application performance is verified by using the traditional setup, where computation uses the approximate units and the application itself is unaltered. During training, inputs enter an approximate branch and an accurate branch. The accurate branch keeps the original application coefficients and produces precise results. The approximate branch accurately models the approximate hardware while using flexible coefficients. The difference between the two branches is then used as the error to train the coefficients in the approximate branch. Training is performed using an optimization solver. The optimizer used will depend on the size of the application kernel and the nature of the approximate computing unit involved (e.g., integer vs. floating point).

## 7.3 Evaluation on Fixed Hardware

### 7.3.1 Approximate hardware

We choose to study approximate multipliers since they add the most energy and time delay costs, as compared to the other arithmetic operations. The multipliers we use are summarized in Table 7.1. We use a subset of multipliers from the EvoApprox library[MHV17a] since the well-defined error metrics provided a clear baseline for comparing their performance in different applications. We also demonstrate improvements when using more widely used multipliers that were intentionally designed for high performance - the error-tolerant multiplier (ETM) [KWK10] and the Dynamic Range Unbiased Multiplier (DRUM) [HBR15]. We

Table 7.1: Multiplier summary. Performance numbers are normalized to 16-bit multipliers.

| Multiplier | Variant | Area | Power |
|---|---|---|---|
| ETM [KWK10] | 8-bit | 0.14 | 0.04 |
| | 16-bit | 0.50 | 0.25 |
| DRUM [HBR15] | 16-bit-4 | 0.25 | 0.12 |
| | 16-bit-6 | 0.39 | 0.29 |
| EvoApprox [MHV17a] | mul8u_JV3 | 0.03 | 0.02 |
| | mul8u_FTA | 0.07 | 0.04 |
| | mul8u_185Q | 0.13 | 0.09 |
| | mul8s_1KR3 | 0.07 | 0.02 |
| | mul8s_1KVL | 0.21 | 0.12 |
| | mul16s_GK2 | 1.01 | 0.89 |
| | mul16s_GAT | 0.74 | 0.58 |

use an 8-bit ETM with the bits split at k = 4 and a 16-bit ETM with the bits split at k = 8 and use two implementations of the 16-bit DRUM with k = 4 and k = 6. Area and power numbers in Table 7.1 are normalized to accurate 16-bit multipliers.

### 7.3.2 Applications

Table 7.2: Application summary

| Application | Coefficients |
|---|---|
| Gaussian blur | 3x3 |
| Edge detection | 3x3 |
| Image sharpening | 3x3 |
| Discrete Cosine Transform | 8x8 |
| Discrete Fourier Transform | 12x12(complex) |
| Inversek2j [YME17] | 4 |

Table 7.2 summarizes the applications used for evaluating LAC. Performance is first evaluated for three applications using 3x3 filters. The three filters include the 3x3 versions of Gaussian blur for image blurring, Sobel filter for edge detection and Laplacian filter [YME17] for image sharpening. Gaussian blur uses unsigned values, so the unsigned multipliers are used for the experiments, while the other two use signed values in the filters. For image sharpening using the Laplacian filter, the outputs of the filter are scaled to $[0, 255]$ and then added to the original image for the final result. Average Structural Similarity Index (SSIM)[WBS04] is used to measure the performance before and after training using LAC since the applications produce image outputs.

To analyze the performance of more complicated applications, we also train the Discrete Cosine Transform (DCT) and the Discrete Fourier Transform (DFT). The DCT uses a quality level of 50, as described in [CG], and requires an 8x8 filter. We initially tried training the DFT using the same 32x32 CIFAR-10 images, but the training ran into runtime and convergence issues. The size of the DFT had to be reduced to 12x12 for the algorithm to finish properly. The scalability of our approach beyond the problem sizes demonstrated here might be addressed in future studies. Both DCT and DFT contain floating-point coefficients, so the coefficients are scaled up by $2^m$ and then rounded to fill the integer input range. The final values are scaled down by the $2^m$ for DCT and $2^{2m}$ for DFT since DFT is performed twice on the x and y axes. The quality of DCT and DFT are measured using the peak signal-to-noise ratio (PSNR) between outputs using approximate multipliers and outputs. Coefficients are constrained to $[0, 2^m - 1]$ for applications using unsigned values, and $[-(2^m - 1), (2^m - 1)]$ for signed values, where $m$ is 8 for the first three applications and 16 for the rest.

We also include Inversek2j from AxBench [YME17] as an application that does not operate on image inputs. Inversek2j computes the inverse kinematics for a 2-joint arm, which is useful in robotic applications. The quality of Inversek2j is measured using relative error. For SSIM and PSNR, a higher value indicates a better quality, while the opposite is true for

relative error.

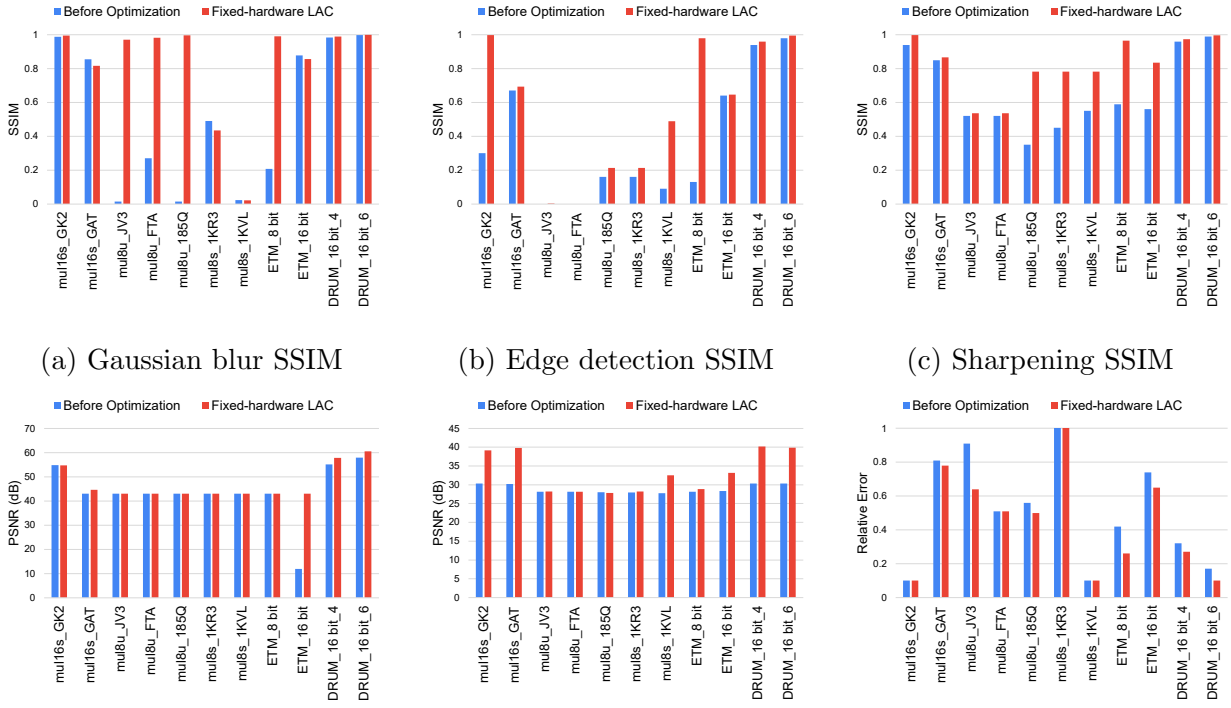
### 7.3.3 Dataset

We use the CIFAR-10 dataset [Kri09] as the input for all of the image applications. Models are trained on 100 training images, and evaluated on the 20 test images.

### 7.3.4 Optimization solvers

The optimization problem was framed as pure integer optimization - for the blurring, edge detection, and sharpening - since in these cases all the variable weights are constrained to being integers by the input requirements of the multipliers. In the case of the DFT and DCT, weights are scaled to force them into integers. These integer or range constraints were also provided to the optimizer.

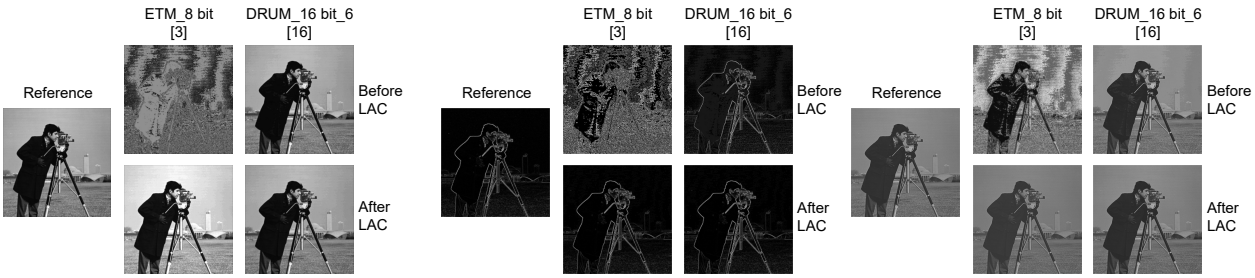
The initial LAC implementation is performed using the Matlab surrogate solver. The Matlab optimizer used is a gradient-free optimizer that works well for applications with a relatively small number of coefficients, but it runs into runtime issues for larger applications. To resolve the runtime issue and allow easier integration with the search component in LACS, we migrated to the gradient-based Adam optimizer in PyTorch. During the training process, we keep a high-precision floating-point copy of the weights and quantize the weights to integers on the fly, similar to the straight-through estimator [Ben13] used for training quantized neural networks. To further speed up the simulation of the approximate computing hardware, we implement parallel versions of the approximate multipliers to spread the work across multiple CPU cores.




(a) Gaussian blur SSIM (b) Edge detection SSIM (c) Sharpening SSIM

(d) DCT PSNR (e) DFT PSNR (f) Inversek2j RAE

Figure 7.3: Quality improvements of (a) Gaussian blur, (b) edge detection, (c) image sharpening, (d) JPEG compression using DCT, (e) DFT, and (f) Inversek2j.




(a) Gaussian blur (b) Edge detection (c) Laplacian sharpening

Figure 7.4: Quality improvements of (a) Gaussian blur, (b) edge detection, and (c)image sharpening. Figures on the right use Pareto optimal multipliers with the highest SSIM before optimization.

### 7.3.5 Results

#### 7.3.5.1 Application quality improvement

Figure 7.3 shows the quality improvements from LAC. While some applications originally use signed parameters, we use both unsigned and signed multipliers for all applications as even unsigned multipliers can benefit from LAC. As is mentioned in Section 7.3.2, Gaussian blurring, edge detection, and image sharpening use SSIM for training and evaluation, while DCT and DFT use PSNR. On average, SSIM improves by 0.30, 0.19, and 0.16 for the three applications.[2] PSNR improves by 3.30dB and 4.38dB for DCT and DFT respectively. For Inversek2j, the relative error is reduced by 0.073 on average. Quality improvements from LAC depend on the application and characteristics of the approximate multipliers. Some multipliers with lower quality before training happen to have large errors when using the original coefficients in the application. For those multipliers, LAC is able to avoid the high-error points in those multipliers and achieve higher quality. The improvement is dramatic in many cases, making previously unusable approximate hardware acceptable.

#### 7.3.5.2 Hardware efficiency impact of LAC

The improved quality from LAC results in a better quality-performance tradeoff for all the applications. Fig. 7.4 compares the output quality before and after LAC optimization for two approximate multipliers. While the results before optimization favor the more expensive approximate multipliers, results after LAC optimization are much closer and allow us to use the cheaper and less accurate multipliers.

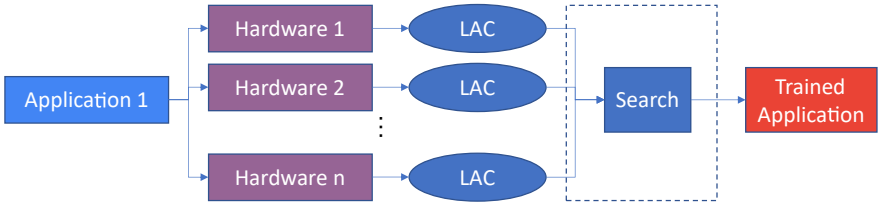



Figure 7.5: Overview of LAC for trained hardware.

## 7.4 Trained Hardware LAC

While training the application for a fixed hardware configuration enables hardware reuse between different applications, it is overly restrictive when designing hardware for an application. For hardware-application co-optimization, we allow searching between hardware configurations. For applications that can choose between different approximate computing hardware, LAC automatically chooses the optimal configuration for a given quality or performance constraint.

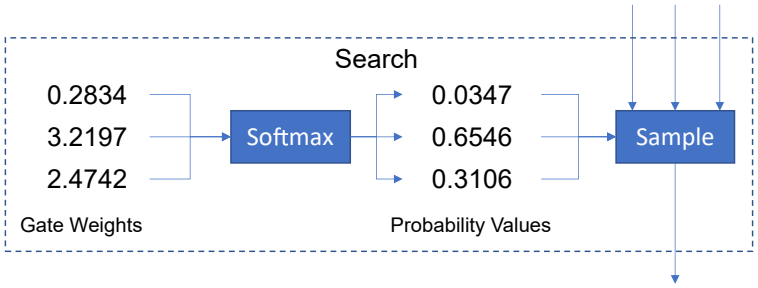



Figure 7.6: Illustration of the binarized gate used for choosing between different hardware outputs.

Fig. 7.5 illustrates the overall structure of LAC for a trained hardware scenario. The application training component uses the same structure as LAC for fixed hardware, which trains the coefficients of an application to minimize the difference between the outputs from approximate hardware and accurate hardware. Multiple approximate hardware is trained at the same time, each with its own set of trainable parameters. The selector at the end

[2]Note that SSIM lies between 0 and 1 with 1 being best.

decides the one to choose that maximizes application performance. The optimization goals here are similar to a neural architecture search (NAS) setup. In neural architectures search, the search algorithm needs to find the optimal network architecture parameters under some constraints from several different options. To apply NAS methods to LAC, we need to replace the network architecture options with approximate hardware options.

To this end, we use the Binary Gate proposed in ProxylessNas [CZH18]. Fig. 7.6 demonstrates the structure of the binarized gate with three inputs, which acts as the "Search" component in Fig. 7.5. In the original ProxylessNas setup, each binary gate is used to choose between multiple layer alternatives for a layer. In LAC, we use it to orchestrate between different approximate multipliers. During training, the binary gate is initialized with the same weight value assigned to each path, where each path represents the application's output using a specific approximate multiplier. In the forward pass, the weight values go through a Softmax function to convert the weight values into probability values. The probability values are used to sample one of the paths in each cycle, The sampled output is then used to calculate the loss. In the backward pass, the application coefficients and the binary gate weights are treated differently. For the application coefficients, the output loss is backpropagated to the sampled path to update the coefficients of the corresponding multiplier. For the binarized gate coefficients, the output loss is backpropagated to all the weight values weighted by the probability values. After training, the path with the highest corresponding weight value in the binarized gate becomes the chosen approximate hardware configuration.

We choose the binarized gate setup for the following reasons:

- Performance. The binarized gate is a point-wise function. and has a low computation cost regardless of the complexity of the applications.
- Scalability. While not extensively explored in this work, the binarized gate setup allows scaling to more complicated applications that can be divided into multiple stages.

For an application with $n$ stages and each stage can choose from $k$ different approximate computing hardware configurations, there are $nk$ distinct options to choose from. Searching for the optimal hardware while training the application coefficients would incur a $nk$ time penalty compared to only training the coefficients. On the other hand, the binarized gate reduces that penalty to $k$ times, since different stages are separated and are controlled by separate binarized gates.

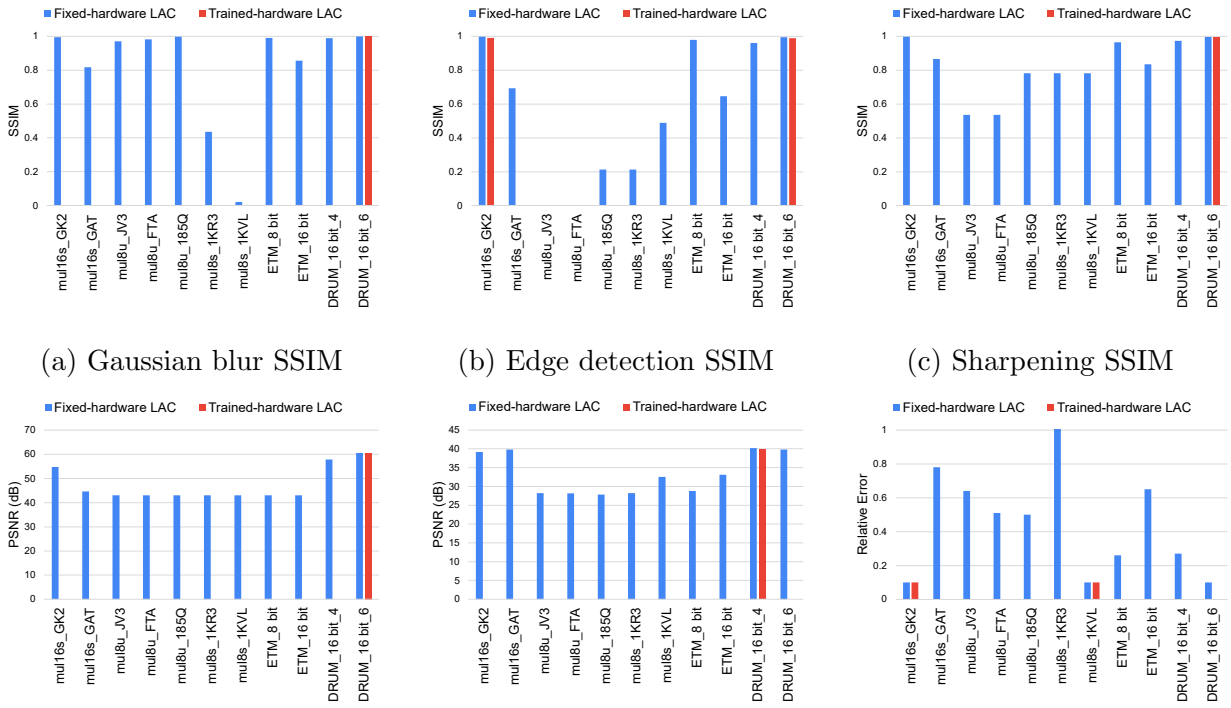
During training, we use Eq. 7.1 to calculate the loss during training. $x$ is the input value, $w$ is the application coefficients to be trained, and $w_0$ is the original application coefficients to calculate the target for optimization. $f(x, w_0)$ calculates the results from accurate hardware using the original coefficients, and $f_a(\cdot)$ is the function to simulate the approximate computing hardware. $b(\cdot)$ then selects one of the possible choices based on the current weights in the binary gate. $L(\cdot)$ represents the output quality of the application and can be peak signal-to-noise ratio (PSNR), structure similarity index measure (SSIM), or something else. As the target applications in our experiments have small memory footprints, we do not use the memory-saving technique in the original ProxylessNas, which samples two paths out of all the paths during each forward path and only updates the parameters corresponding to the two paths. Instead, we train the paths corresponding to all the approximate multiplier options simultaneously.

$$L_a(x, w, t) = L(b(f_a(x, w)), f(x, w_0)) \tag{7.1}$$

The loss performance in Eq. 7.1 only searches for optimal accuracy without performance considerations. For cases where there is a performance constraint (area/power), we reduce the search space to only include multipliers that satisfy the performance constraint. This approach is different from the original ProxylessNas setup, which uses the performance constraint as a regulatory loss term. In a multi-layer neural network, it is not practical to remove all the options violating the performance constraint without being too restrictive on

the search space. This is not an issue in the current LAC setup, where the search process needs to find a single approximate multiplier setup for all relevant multiplications in the application. Removing multipliers that dissatisfy performance constraints also reduces the search space and training time.

## 7.5 Evaluation on Trained Hardware




(a) Gaussian blur SSIM (b) Edge detection SSIM (c) Sharpening SSIM

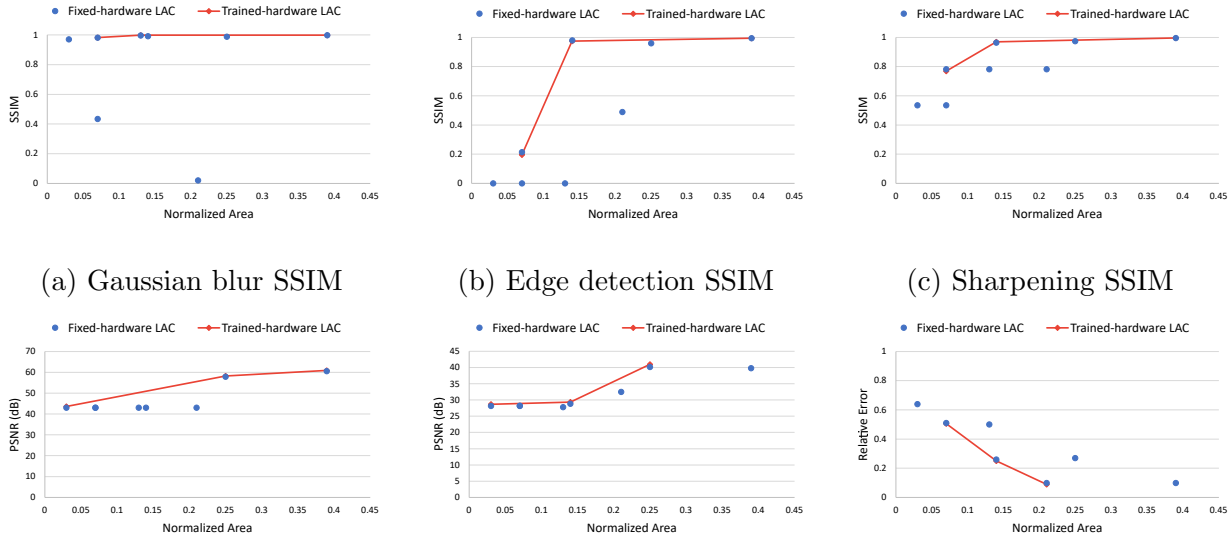
(d) DCT PSNR (e) DFT PSNR (f) Inversek2j RAE

Figure 7.7: LAC search results of (a) Gaussian blur, (b) edge detection, (c) image sharpening, (d) JPEG compression using DCT, (e) DFT, and (f) Inversek2j.

We use the same set of approximate multipliers as Sec. 7.3 to evaluate the performance of LAC when performing a hardware search at the same time.

#### 7.5.0.1 Quality search results

Fig. 7.7 compares the output quality with and without hardware training enabled. For all of the tested applications, LAC is able to find the approximate computing setup that has the highest quality after training. Despite training for multiple multipliers at the same time, LAC does not degrade the performance of the best-performing multiplier significantly.




(a) Gaussian blur SSIM (b) Edge detection SSIM (c) Sharpening SSIM

(d) DCT PSNR (e) DFT PSNR (f) Inversek2j RAE

Figure 7.8: LAC performance-centric search results of (a) Gaussian blur, (b) edge detection, (c) image sharpening, (d) JPEG compression using DCT, (e) DFT, and (f) Inversek2j.

#### 7.5.0.2 Performance search results

To evaluate the search performance of LAC under a performance constraint, we tested the performance of LAC with an area constraint (power constraints generate similar results). We limit the area constraint to 0-0.4x the area of a 16-bit multiplier, as the DRUM multiplier in the 6-bit setup (DRUM_16 bit_6) achieves almost perfect accuracy at $\approx 0.4x$ the size of an accurate multiplier. Fig. 7.8 shows the quality-performance tradeoff with an area constraint. The training algorithm is able to find the approximate hardware with the highest output

quality regardless of the area constraint.

## 7.6 Conclusion

Approximate arithmetic units are a promising method to reduce energy, cost, and latency in error-tolerant applications. So far, research has focused on the optimization of the hardware approximation to minimize application quality loss. In this work, we flip the argument and propose the learned approximate computing approach where the application kernels are optimized to improve application quality in the presence of approximate hardware. In a fixed-hardware setup, We have shown improvements of 0.22 in SSIM and 3.84db in PSNR on average across a variety of signal/image processing applications, showing the utility of LAC as an approach to making the use of approximate hardware more broadly viable. In a trained-hardware setup, LAC allows searching for the best hardware configuration during training. With a method inspired by works on neural architecture search, LAC can find the best approximate hardware configuration both with and without a performance constraint.

# CHAPTER 8

# Conclusion

With the rising demand for machine learning using deep neural networks, this dissertation tries to improve the efficiency of neural network inference using stochastic computing and proposed multiple methods to improve the viability of SC-based neural networks. The contributions are summarized as follows:

- Chapter 2 discusses 3pxnet, which attempts to combine binarization with model pruning to achieve the maximal model compression. Combining the two approaches is not trivial due to the indexing overhead and accuracy concerns. The 3px approach with pruning, permutation, and packing enables meaningful memory and computation reductions over both binarized and pruned neural networks and showcases better accuracy-performance tradeoffs. Despite the improvements, performance and accuracy limitations inherent to binarized neural networks lead to the consideration of stochastic computing.

- Chapter 3 introduces a baseline implementation of an SC-based neural network, which is used in ACOUSTIC [RLM20]. Split-unipolar stream representation improves stream representation accuracy. Average pooling with computation skipping improves effective accumulation precision. OR-based accumulation avoids expensive accumulation in binarized networks. These setups form the basis of SC-based neural networks and showcase their potential performance benefits over traditional fixed-point accelerators.

- Chapter 4 introduces GEO [LRP21], which improves upon ACOUSTIC by optimizing

the stream generation and execution components of stochastic computing. On the stream generation front, maximal-length LFSR ensures accurate stream representation and smart sharing of LFSR seeds reduces training difficulties. On the execution front, partial binary accumulation improves the range precision of SC accumulation with minimal overheads over OR-based accumulation. These two techniques combined reduce stream length requirement by 4X while improving accuracy.

- Chapter 5 introduces REX-SC, mainly aimed to resolve the training difficulties introduced in GEO. Compared to partial binary accumulation, the introduced OR_n methodology enables better model trainability and more effective use of the increased accumulation range. These benefits allow us to demonstrate accurate data for more complicated models and datasets.

- Chapter 6 introduces training optimizations for stochastic computing, which also proves to be effective for other approximate computing methods. The simulation improvements allow efficient simulation of SC computation on Nvidia GPUs and reduce the overhead of stream simulation. The overhead is further minimized by replacing stream simulation with error injection training for most of the epochs. During backpropagation, activation proxies approximate the effect of non-linear accumulations and allow the usage of optimized kernels for normal convolutional and linear layers. These optimizations significantly reduce the training overhead of neural networks using stochastic computing and other approximate computing methods.

- Chapter 7 introduces LAC, which aims to extend training for approximate computing to applications other than neural networks. We show that an almost arbitrary parameterized application can be trained to work better with approximate computing. Training the application reduces the accuracy gap between high-accuracy and high-performance approximate multipliers and allows the reuse of approximate computing hardware across multiple applications. We also tried to search for the optimal approx-

imate computing hardware when the hardware is not fixed. By applying techniques inspired by neural architecture search, we can find the optimal hardware configuration while also training the application for the hardware.

While this dissertation proposes methods to improve the viability of SC-based neural networks, there are still areas for improvement and further research, including:

## 8.1 Validation of SC-based Neural Networks on More Models and Datasets

Most of the work in this thesis focuses on convolutional neural networks on vision-type datasets. There have been attempts to measure the accuracy of transformer-style models and other types of data, including speech and text. However, the high training cost of transformer models prevented this from happening within the scope of this thesis.

The difficulty in training mainly comes from the need for pretraining for transformer models. Take the BERT-large model for instance. BERT-large is one of the MLPerf training benchmarks, and from the results, the time to converge is similar to Resnet-50 training on ImageNet. While Resnet-50 results are not discussed in this thesis, it takes 4-5 days to train to completion using REX-SC after the training optimizations. This is long, but still doable. The problem is that the BERT-large training task is for fine-tuning only. Due to the large parameter size, transformer models often require extensive pre-training on unlabeled data, which is much more expensive than the fine-tuning phase. The difficulty of fine-tuning can be seen in the research work surrounding the recently released LLaMa language model from Meta. Most of the works focus on fine-tuning the provided weights, as pre-training a multi-billion-parameter model is prohibitively expensive for a lot of people. As discussed in Chap. 6, SC cannot directly use models pre-trained for floating-point or fixed-point computation, meaning that verifying SC accuracy on transformer models requires pretraining as well.

Future work to validate SC accuracy on transformer models can consider the following options:

- Perform fine-tuning in multiple stages. Floating-point models cannot be transferred to REX-SC using OR_2-4 due to the non-linearity introduced by OR_n. To get around this issue, the floating-point model can be first fine-tuned for SC using binary accumulation, which is equivalent to OR_inf. After that, the n value in OR_n can be reduced successively to finally reach the desired value.

- Remove/reduce dependence on pretraining. Transformer models require pretraining due to the lack of inductive bias, which is particularly prevalent in vision transformers [DBK20] when compared to convolutional neural networks. If transformer models can be modified to better match the target architecture, it may be possible to reduce or remove the pretraining stage of training.

## 8.2 Efficient Model Transfer for Stochastic Computing

Despite the improvements made to the training algorithm for SC, training is not always an option in real-life applications. Compared to SC which still requires dedicated training, 8-bit fixed-point quantization is often possible without access to labeled training data. Similar to the previous problem, this problem (partly) stems from the non-linearity of OR_n accumulation. To resolve this issue, there are several possible options:

- Reduce the non-linearity of SC accumulation. The current SC accumulation methods need to choose between linearity (binary accumulation) and performance (partial binary and OR_n accumulation), while the method that satisfies both (mux accumulation) suffers from scaling issues. Future optimizations to SC computation should strive to achieve linear accumulation while maintaining the performance benefits of SC accumulation.

- Reduce the reliance on linear accumulation. Accumulation is assumed to be accurate for most works on fixed-point quantization, and that assumption results in difficulties when transferring a model to non-linear accumulation. The assumption of linear accumulation may not be essential, and future works can examine the importance of linear accumulation and explore methods to bypass this assumption.

## 8.3 Enabling SC for Large Language Models

The release of ChatGPT brought increased interest to large language models (LLM), and SC seems like a good candidate for LLM given the high computation requirements. LLMs are usually based on transformer models, so the previous discussions on model training and transfer also apply here. Future work on SC for LLM should focus on transferring pretrained LLM to SC without extensive training.

Even if model transfer is made possible for SC, additional work may be needed for the fine-tuning process. Pre-trained LLMs often need to be fine-tuned when used for a particular downstream task. Given the large parameter size of LLMs (ranging from several billion to more than a trillion), it can be unfeasible to perform fine-tuning with gradient descent on all parameters. Instead, recent works like LoRA[HSW21] perform fine-tuning on a reduced parameter set to save memory. Future work on SC should allow alternative fine-tuning algorithms to improve the flexibility of SC.

## 8.4 SC for Homomorphic Encryption

Homomorphic encryption preserves data security by performing computation on encrypted data without access to the encryption key. Stochastic computing represents numbers in an alternative format. Since SC computation can be highly-efficient in the SC domain with single-gate multiplications and additions, it can be useful for homomorphic encryption.

However, several factors need to be considered:

- Data security. SC bit streams are not encrypted by default. Converting SC streams to fixed-point numbers does not require an encryption key and can be done using just counters. As such, both conversions to and from SC need to be modified to ensure data security while preserving the bit-wise compute units in SC.
- Error tolerance. Stochastic computing is random in nature and can introduce random errors in the computation results. The encryption scheme should be designed such that small errors in the encrypted format result in small to no errors in the decrypted format.

## 8.5 Specialized SC Architectures for Emerging Compute Paradigms

The improvements to stochastic computing mostly center around the CMOS implementation of SC. While Chap. 5 discusses the implementation of REX-SC for compute-in-memory (CIM) systems, REX-SC is not initially designed with CIM in mind. As a result, better performance and accuracy can be achieved by tailoring the SC algorithm to CIM properties. The following are several options that show promising results.

### 8.5.1 Range-adjustable OR_n accumulation

The OR_n exploration in Chap. 5 focus on finding OR_n gates that are limited to n as the maximal output value. This limitation makes sense for the CMOS implementation but is not necessary for the CIM implementation. Fig. 8.1 demonstrates an OR_1 accumulation behavior in CIM. In contrast to CMOS SC which needs to limit the precision of intermediate accumulation to ensure performance, CIM SC maintains a high-precision accumulation result before the ADC, and there is no need to limit the accumulation range before ADC. In the case of OR_n, this means that the threshold of the n output bits can be arbitrarily set with

little effect on performance.

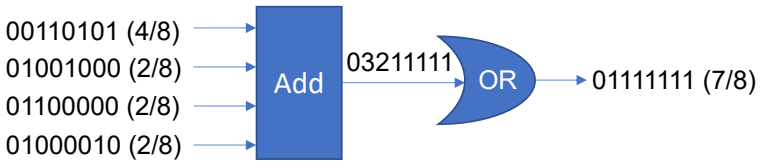



Figure 8.1: Accumulation demonstration of OR-based SCIM accumulation

In preliminary experiments, range-adjustable OR_n can roughly reduce n by 1 compared to fixed-range OR_n. This means range-adjustable OR_1 has comparable accuracy to OR_2 without range adjustment. Despite the potential benefits, future work should carefully compare the cost saving of one fewer output bit with the additional cost of adding range adjustment. As the range adjustment needs to happen during inference, the hardware needs to support range switching on the fly, which may become expensive.

### 8.5.2 Delta-sigma SC

Range-adjustable OR_n tries to make the most out of the high-precision pre-ADC value in CIM arrays, but such an approach can become limited by the randomness of SC. In the example shown in Fig. 8.1, only two out of the 8 cycles require ADC to distinguish more than just OR_1. In this example, OR_3 would be required to completely avoid information loss at the ADC, but the additional output bits are completely wasted in other cycles. With these limitations in mind, an improved ADC for SCIM should satisfy the following two constraints:

- Low output precision. The ADC should generate only a few output bits. Since SC requires the generation of outputs over multiple cycles, forcing the ADC to generate too many bits per cycle impacts its effectiveness.
- Remainder storage. If the accumulation result of the current cycle cannot be accurately represented by the output value, the quantization error should be stored so that it can

(potentially) be corrected in a future cycle.

The first constraint is naturally satisfied by an OR-based adder, but the second constraint enforces some type of storage of the accumulation result between cycles. One possible implementation that satisfies this constraint while limiting the output precision to 1 bit/cycle comprises three components: accumulator, remainder capacitor, and subtractor. In each cycle, the accumulator adds the accumulation result onto the remainder capacitor. If the value stored in the capacitor is greater or equal to a set threshold $T$, it outputs 1 (interpreted as T) and the subtractor reduces the capacitor value by $T$. If the value of the capacitor is smaller than $T$, it outputs 0 and keeps the capacitor value. For streams with average values smaller than $T$, the result will be accurate up to the size of the remainder storage. For streams with average values larger than $T$, the result will be saturated at $T$. This saturation behavior is easy to model during training, and the ADC achieves relatively accurate accumulation for results between 0 and $T$. The end result looks very similar to a sigma-delta modulator and also retains its noise-shaping properties.

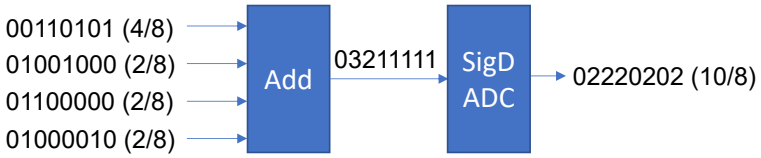



Figure 8.2: Accumulation demonstration of SD SCIM accumulation

Fig. 8.2 shows how a sigma-delta ADC works in SCIM. Compared to OR-based SCIM, the sigma-delta (SD) version completely avoids information loss when $T = 2$. Although the figure shows the ADC outputting values up to 2, it only outputs two values, which can be represented using a single bit. Compared to using a high-precision Successive-approximation-register (SAR) ADC, the sigma-delta version only outputs a single bit per cycle.

With $T = 8$ and a maximal accumulated value of 15, SC with sigma-delta ADC (DSM-SC) is competitive with SC using accurate binary accumulation and when compared with a 6-bit fixed-point baseline, as shown in Fig. 8.3. Despite only outputting a single bit after

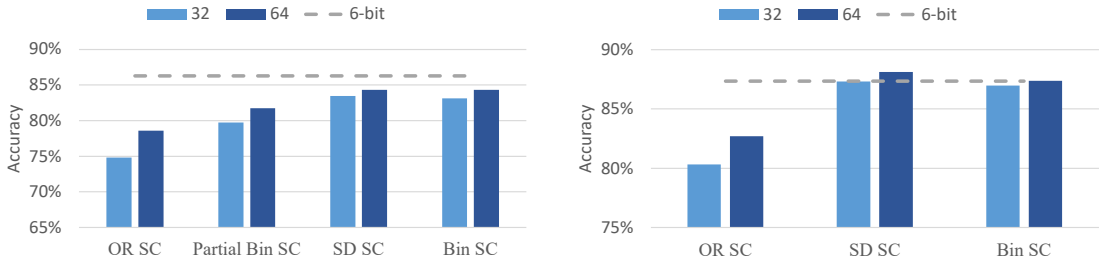



Figure 8.3: Accuracy of SC with sigma-delta ADC compared to OR SC and SC with binary accumulation for (a) TinyConv on CIFAR-10 and (b) Resnet-18 on ImageNet.

each cycle, the error-shaping behavior of the sigma-delta modulator limits the overall error and achieves high overall accuracy.

Despite the promising accuracy results, several factors must be considered when using DSM-SC. The first is that DSM-SC focuses on small output values. Maintaining high accuracy for small values is expensive for CIM. The second is the accuracy results in Fig. 8.3 assumes that the "sigma" component in DSM is accurate. Non-scaled addition is more expensive than scaled addition, similar to the trade-offs for stochastic addition. Preliminary experiments show that replacing the non-scaled addition with scaled addition significantly reduces the accuracy benefits of DSM-SC, and future work on this approach needs to consider this tradeoff.

# REFERENCES


[ACR18] Renzo Andri, Lukas Cavigelli, Davide Rossi, and Luca Benini. "Yoda NN: An architecture for ultralow power binary-weight CNN acceleration." *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, **37**(1):48–60, 2018.

[AH13] Armin Alaghi and John P Hayes. "Survey of Stochastic Computing." *ACM Transactions on Embedded computing systems (TECS)*, **12**(2s):1–19, 2013.

[AHS17] Sajid Anwar, Kyuyeon Hwang, and Wonyong Sung. "Structured Pruning of Deep Convolutional Neural Networks." *ACM Journal on Emerging Technologies in Computing Systems*, **13**(3):1–18, 2017.

[ALP17] Hande Alemdar, Vincent Leroy, Adrien Prost-Boucle, and Frederic Petrot. "Ternary neural networks for resource-efficient AI applications." *Proceedings of the International Joint Conference on Neural Networks*, **2017-May**:2547–2554, 2017.

[ARM18] ARM. "ARM Compute Library.", 2018.

[ARM19] ARM. "ARM CMSIS-NN.", 2019.

[AS16] Sajid Anwar and Wonyong Sung. "Compact Deep Convolutional Neural Networks With Coarse Pruning." *arXiv preprint arXiv:1610.09639*, 2016.

[Bav] Louis Bavoil. "Optimizing Compute Shaders for L2 Locality using Thread-Group ID Swizzling — NVIDIA Technical Blog.".

[BC18] 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 *ISSCC*, pp. 488–490, 2018.

[Ben13] Yoshua Bengio. "Estimating or Propagating Gradients Through Stochastic Neurons." *CoRR*, **abs/1305.2982**, 2013.

[BH22] Timothy J. Baker and John P. Hayes. "CeMux: Maximizing the Accuracy of Stochastic Mux Adders and an Application to Filter Design." *ACM Trans. Des. Autom. Electron. Syst.*, **27**(3), jan 2022.

[BKA18] Andrawes Al Bahou, Geethan Karunaratne, Renzo Andri, Lukas Cavigelli, and Luca Benini. "XNORBIN: A 95 TOp/s/W Hardware Accelerator for Binary Convolutional Neural Networks." *CoRR*, **abs/1803.0**, 2018.

[BNS19] Ron Banner, Yury Nahshan, and Daniel Soudry. "Post training 4-bit quantization of convolutional networks for rapid-deployment." In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[BRT21] Colby Banbury, Vijay Janapa Reddi, Peter Torelli, Jeremy Holleman, Nat Jeffries, Csaba Kiraly, Pietro Montino, David Kanter, Sebastian Ahmed, Danilo Pau, et al. "Mlperf tiny benchmark." *arXiv preprint arXiv:2106.07597*, 2021.

[BTF17] A. Bonetti, A. Teman, P. Flatresse, and A. Burg. "Multipliers-Driven Perturbation of Coefficients for Low-Power Operation in Reconfigurable FIR Filters." *TCAS1*, **64**(9):2388–2400, 2017.

[BYM18] Daniel Bankman, Lita Yang, Bert Moons, Marian Verhelst, and Boris Murmann. "An Always-On 3.8µJ/86% CIFAR-10 Mixed-Signal Binary CNN Processor with All Memory on Chip in 28nm CMOS." In *2018 IEEE International Solid - State Circuits Conference - (ISSCC)*, pp. 222–224, 2018.

[CCR13] Vinay K Chippa, Srimat T Chakradhar, Kaushik Roy, and Anand 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.

[CDS90] Yann Le Cun, John S Denker, and Sara a Solla. "Optimal Brain Damage." *Advances in Neural Information Processing Systems*, **2**(1):598–605, 1990.

[CG] Ken Cabeen and Peter Gent. "Image Compression and the Discrete Cosine Transform.".

[CHS16] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. "Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1." *CoRR*, **abs/1602.0**, 2016.

*[Supposedly first BNN with both weights and inputs binarized.]*

[CKE16] Y H Chen, T Krishna, J Emer, and V Sze. "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks." *Proc. ISSCC*, **52**(1):262–263, 2016.

[CLL15] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. "MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems." *eprint arXiv:1512.01274*, pp. 1–6, 2015.

[CLX16] Ping Chi, Shuangchen Li, Cong Xu, Tao Zhang, Jishen Zhao, Yongpan Liu, Yu Wang, and Yuan Xie. "PRIME: A Novel Processing-in-Memory Architecture for Neural Network Computation in ReRAM-Based Main Memory." *SIGARCH Comput. Archit. News*, **44**(3):27–39, jun 2016.

[CSB18] Francesco Conti, Pasquale Davide Schiavone, and Luca Benini. "XNOR Neural Engine: a Hardware Accelerator IP for 21.6 fJ/op Binary Neural Network Inference." *ArXiv e-prints*, **abs/1807.0**, 2018.

[CSM15] Zhiyong Cheng, Daniel Soudry, Zexi Mao, and Zhenzhong Lan. "Training Binary Multilayer Neural Networks for Image Classification using Expectation Backpropagation." *CoRR*, **abs/1503.0**, 2015.

[CVR14] Vinay K. Chippa, Swagath Venkataramani, Kaushik Roy, and Anand 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.

[CWZ17] Yu Cheng, Duo Wang, Pan Zhou, and Tao Zhang. "A Survey of Model Compression and Acceleration for Deep Neural Networks." *CoRR*, **abs/1710.0**:1–10, 2017.

[CXZ16] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. "Training Deep Nets with Sublinear Memory Cost." *CoRR*, **abs/1604.06174**, 2016.

[CZH18] Han Cai, Ligeng Zhu, and Song Han. "ProxylessNAS: Direct Neural Architecture Search on Target Task and Hardware." *CoRR*, **abs/1812.00332**, 2018.

[DBK20] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale." *CoRR*, **abs/2010.11929**, 2020.

[DCL18] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding." *CoRR*, **abs/1810.04805**, 2018.

[DJP18] Lei Deng, Peng Jiao, Jing Pei, Zhenzhi Wu, and Guoqi Li. "GXNOR-Net: Training deep neural networks with ternary weights and activations without full-precision memory under a unified discretization framework." *Neural Networks*, **100**:49–58, 2018.

[DM98] L. Dagum and R. Menon. "OpenMP: an industry standard API for shared-memory programming." *IEEE Computational Science and Engineering*, **5**(1):46–55, 1998.

[DMC93] Jeffrey A Dickson, Robert D Mcleod, and Howard C Card. "Stochastic arithmetic implementations of neural networks with in situ learning." In *IEEE International Conference on Neural Networks*, pp. 711–716 vol.2, 1993.

[FAF13] F. Farshchi, M. S. Abrishami, and S. M. Fakhraie. "New approximate multiplier for low power digital signal processing." In *CADS 2013*, pp. 25–30, 2013.

[FFG17] Julian Faraone, Nicholas Fraser, Giulio Gambardella, Michaela Blott, and Philip H.W. Leong. "Compressing Low Precision Deep Neural Networks Using Sparsity-Induced Regularization in Ternary Networks." *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, **10635 LNCS**:393–404, 2017.

[FTP15] Hassan Foroosh, Marshall Tappen, and Marianna Penksy. "Sparse Convolutional Neural Networks." *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 806–814, 2015.

[FUG17] Nicholas J. Fraser, Yaman Umuroglu, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees 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*, PARMA-DITAM '17, p. 25–30, New York, NY, USA, 2017. Association for Computing Machinery.

[GAG15] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. "Deep Learning with Limited Numerical Precision." 2015.

[Gai67] Brian R Gaines. "Stochastic computing." In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pp. 149–156, 1967.

[GGS18] Patricia Gonzalez-Guerrero, Xinfei Guo, and Mircea Stan. "SC-SD: Towards Low Power Stochastic Computing Using Sigma Delta Streams." In *2018 IEEE International Conference on Rebooting Computing (ICRC)*, pp. 1–8, 2018.

[GH17] Mingyu Gao and Mark Horowitz. "TETRIS: Scalable and Efficient Neural Network Acceleration with 3D Memory." In *ASPLOS*, pp. 751–764, 2017.

[GLG22] Vaibhav Gupta, Tianmu Li, and Puneet Gupta. "LAC: Learned Approximate Computing." In *IEEE/ACM Design, Automation and Test in Europe*, p. 4, March 2022.

[GSG17] Chirag Gupta, Arun Sai Suggala, Ankit Goyal, Harsha Vardhan Simhadri, Bhargavi Paranjape, Ashish Kumar, Saurabh Goyal, Raghavendra Udupa, Manik Varma, and Prateek Jain. "ProtoNN: Compressed and Accurate kNN for

Resource-scarce Devices." *34th International Conference on Machine Learning (ICML 2017)*, **70**:1331–1340, 2017.

[GSG22] Jing Gong, Hassaan Saadat, Hasindu Gamaarachchi, Haris Javaid, Xiaobo Sharon Hu, and Sri Parameswaran. "ApproxTrain: Fast Simulation of Approximate Multipliers for DNN Training and Inference.", 2022.

[HBM22] Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, Tom Hennigan, Eric Noland, Katie Millican, George van den Driessche, Bogdan Damoc, Aurelia Guy, Simon Osindero, Karen Simonyan, Erich Elsen, Jack W. Rae, Oriol Vinyals, and Laurent Sifre. "Training Compute-Optimal Large Language Models.", 2022.

[HBO18] Mattias P. Heinrich, Max Blendowski, and Ozan 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.

[HBR15] Soheil Hashemi, R. Iris Bahar, and Sherief Reda. "DRUM: A Dynamic Range Unbiased Multiplier for approximate applications." In *ICCAD 2015*, pp. 418–425, 2015.

[HGF18] Zhezhi He, Boqing Gong, and Deliang Fan. "Optimize Deep Convolutional Neural Network with Ternarized Weights and High Accuracy." 2018.

[HGT19] Reza Hojabr, Kamyar Givaki, S M Reza Tayaranian, Parsa Esfahanian, Ahmad Khonsari, Dara Rahmati, and M Hassan Najafi. "SkippyNN : An Embedded Stochastic-Computing Accelerator for Convolutional Neural Networks." In *2019 56th ACM/IEEE Design Automation Conference (DAC)*, pp. 1–6. ACM, 2019.

[HMD15] Song Han, Huizi Mao, and William J. Dally. "Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding." pp. 1–14, 2015.

[HPT15] Song Han, Jeff Pool, John Tran, and William J. Dally. "Learning both Weights and Connections for Efficient Neural Networks." pp. 1–9, 2015.

[HS99] R. Hegde and N. R. Shanbhag. "Energy-efficient signal processing via algorithmic noise-tolerance." In *ISLPED 1999*, pp. 30–35, 1999.

[HSW89] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. "Multilayer feedforward networks are universal approximators." *Neural Networks*, **2**(5):359–366, 1989.

[HSW93] Babak Hassibi, David G. Stork, and Gregory J. Wolff. "Optimal brain surgeon and general network pruning.", 1993.

[HSW21] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, and Weizhu Chen. "LoRA: Low-Rank Adaptation of Large Language Models." *CoRR*, **abs/2106.09685**, 2021.

[HZC17] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications." 2017.

[HZL18] Yuwei Hu, Jidong Zhai, Dinghua Li, Yifan Gong, Yuhao Zhu, Wei Liu, Lei Su, and Jiangming Jin. "BitFlow: Exploiting Vector Parallelism for Binary Neural Networks on CPU." *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 244–253, 2018.

[HZR16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian 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.

[IHM16] Forrest N. Iandola, Song Han, Matthew W. Moskewicz, Khalid Ashraf, William J. Dally, and Kurt Keutzer. "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size." pp. 1–13, 2016.

[IIS14] Hideyuki Ichihara, Shota Ishii, Daiki Sunamori, Tsuyoshi Iwagaki, and Tomoo Inoue. "Compact and accurate stochastic circuits with shared random number sources." *ICCD*, pp. 361–366, 2014.

[Int21] Intel. "Intel® Intrinsics Guide.", 2021.

[JAM17] Patrick Judd, Jorge Albericio, and Andreas Moshovos. "Stripes: Bit-Serial Deep Neural Network Computing." *IEEE Computer Architecture Letters*, **16**(1):80–83, 2017.

[JBB17] Norman P. Jouppi, Al Borchers, and Rick et.al. Boyle. "In-Datacenter Performance Analysis of a Tensor Processing Unit." *Proceedings of the 44th Annual International Symposium on Computer Architecture - ISCA '17*, pp. 1–12, 2017.

[JDS17] Patrick Judd, Alberto Delmas, Sayeh Sharify, and Andreas Moshovos. "Cnvlutin2: Ineffectual-Activation-and-Weight-Free Deep Neural Network Computing." pp. 1–6, 2017.

[JGK15] A. Jaiswal, B. Garg, V. Kaushal, and G. K. Sharma. "SPAA-Aware 2D Gaussian Smoothing Filter Design Using Efficient Approximation Techniques." In *VLSID 2015*, pp. 333–338, 2015.

[JKC18] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. "Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference." In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.

[JKW17] Lei Jiang, Minje Kim, Wujie Wen, and Danghui Wang. "XNOR-POP: A processing-in-memory architecture for binary Convolutional Neural Networks in Wide-IO2 DRAMs." *Proceedings of the International Symposium on Low Power Electronics and Design*, 2017.

[KB14] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization.", 2014. cite arxiv:1412.6980Comment: Published as a conference paper at the 3rd International Conference for Learning Representations, San Diego, 2015.

[KBM17] Abhisek Kundu, Kunal Banerjee, Naveen Mellempudi, Dheevatsa Mudigere, Dipankar Das, Bharat Kaul, and Pradeep Dubey. "Ternary Residual Networks." pp. 1–16, 2017.

[KGE11] P. Kulkarni, P. Gupta, and M. Ercegovac. "Trading Accuracy for Power with an Underdesigned Multiplier Architecture." In *VLSID 2011*, pp. 346–351, 2011.

[KGV17] Ashish Kumar, Saurabh Goyal, and Manik Varma. "Resource-efficient Machine Learning in 2 KB RAM for the Internet of Things." *34th International Conference on Machine Learning (ICML 2017)*, **70**:1935–1944, 2017.

[KLC15] K. Kim, J. Lee, and K. Choi. "Approximate de-randomizer for stochastic circuits." In *ISOCC*, pp. 123–124, 2015.

[Kri09] Alex Krizhevsky. "Learning Multiple Layers of Features from Tiny Images." pp. 32–33, 2009.

[KS16] Minje Kim and Paris Smaragdis. "Bitwise Neural Networks." **37**, 2016.

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "ImageNet Classification with Deep Convolutional Neural Networks." In *Communications of the ACM*, pp. 1106–1114, 2012. arXiv: 1102.0183 ISSN: 10495258.

[KWK10] Khaing Yin Kyaw, Wang Ling Goh, and Kiat Seng Yeo. "Low-power high-speed multiplier for error-tolerant application." In *EDSSC 2010*, pp. 1–4, 2010.

[LBB98] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. "Gradient-based learning applied to document recognition." *Proceedings of the IEEE*, **86**(11):2278–2324, 1998.

[LDZ18] Ling Liang, Lei Deng, Yueling Zeng, Xing Hu, Yu Ji, Xin Ma, Guoqi Li, and Yuan Xie. "Crossbar-aware neural network pruning." pp. 1–13, 2018.

[LH17] Siting Liu and Jie Han. "Energy Efficient Stochastic Computing with Sobol Sequences." In *DATE*, pp. 650–653. EDAA, 2017.

[Li18] Yuan Li, Shuangchen and Oliver Glova, Alvin and Hu, Xing and Gu, Peng and Niu, Dimin and T. Malladi, Krishna and Zheng, Hongzhong and Brennan, Bob and Xie. "SCOPE: A Stochastic Computing Engine for DRAM-based In-situ Accelerator." In *IEEE Micro*, pp. 697–710, 2018.

[LL15] Vadim Lebedev and Victor Lempitsky. "Fast ConvNets Using Group-wise Brain Damage." 2015.

[LLG23] Tianmu Li, Shurui Li, and Puneet Gupta. "Training Neural Networks for Execution on Approximate Hardware." In *International Research Symposium on Tiny Machine Learning (tinyML)*, p. 6, March 2023.

[LLR18] Zhe Li, Ji Li, Ao Ren, Ruizhe Cai, Caiwen Ding, Xuehai Qian, Jeffrey Draper, Bo Yuan, Jian Tang, Qinru 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*, **38**(8):1543–1556, 2018.

[LLX17] Yixing Li, Zichuan Liu, Kai Xu, Hao Yu, and Fengbo Ren. "A 7.663-TOPS 8.2-W Energy-efficient FPGA Accelerator for Binary Convolutional Neural Networks (Abstract Only)." *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays - FPGA '17*, (March):290–291, 2017.

[LR18] Yixing Li and Fengbo Ren. "Build a Compact Binary Neural Network through Bit-level Sensitivity and Data Pruning." 2018.

[LRL17] Zhe Li, Ao Ren, Ji Li, Qinru Qiu, Bo Yuan, Jeffrey Draper, and Yanzhi Wang. "Structural design optimization for deep convolutional neural networks using stochastic computing." *Proceedings of the 2017 Design, Automation and Test in Europe, DATE 2017*, (c):250–253, 2017.

[LRP21] 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)*, pp. 1–6, 2021.

[LRP23] Tianmu Li, Wojciech Romaszkan, Sudhakar Pamarti, and Puneet Gupta. "REX-SC: Range-Extended Stochastic Computing Accumulation for Neural Network Acceleration." *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, June 2023.

[LSC18] Liangzhen Lai, Naveen Suda, and Vikas Chandra. "CMSIS-NN: Efficient Neural Network Kernels for Arm Cortex-M CPUs." *CoRR*, **abs/1801.06601**, 2018.

[LXY17] Yixing Li, Kai Xu, and Hao Yu. "A GPU-Outperforming FPGA Accelerator Architecture for Binary Convolutional Neural Networks." *CoRR*, 2017.

[LXZ17] Jeng Hau Lin, Tianwei Xing, Ritchie Zhao, Zhiru Zhang, Mani Srivastava, Zhuowen Tu, and Rajesh K. Gupta. "Binarized Convolutional Neural Networks with Separable Filters for Efficient Hardware Acceleration." *IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, **2017-July**(1):344–352, 2017.

[LZL16] Fengfu Li, Bo Zhang, and Bin Liu. "Ternary Weight Networks." (Nips), 2016.

[LZP17] Xiaofan Lin, Cong Zhao, and Wei Pan. "Towards Accurate Binary Convolutional Neural Network." (3):1–14, 2017.

[MBJ09] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P Jouppi. "CACTI 6.0 : A Tool to Model Large Caches." *HP laboratories*, **27**(HPL-2009-85):28, 2009.

[MCA14] Sasa Misailovic, Michael Carbin, Sara Achour, Zichao Qi, and Martin C. Rinard. "Chisel: Reliability- and Accuracy-Aware Optimization of Approximate Computational Kernels." *SIGPLAN Not.*, **49**(10):309–328, October 2014.

[MHT19] M. Masadeh, O. Hasan, and S. Tahar. "Using Machine Learning for Quality Configurable Approximate Computing." In *2019 DATE*, pp. 1575–1578, 2019.

[MHT20] M. Masadeh, O. Hasan, and S. Tahar. "Machine Learning-Based Self-Compensating Approximate Computing." In *2020 SysCon*, pp. 1–6, 2020.

[MHV17a] V. Mrazek, R. Hrbacek, Z. Vasicek, and L. Sekanina. "EvoApprox8b: Library of Approximate Adders and Multipliers for Circuit Design and Benchmarking of Approximation Methods." In *DATE 2017*, pp. 258–261, 2017.

[MHV17b] Vojtech Mrazek, Radek Hrbacek, Zdenek Vasicek, and Lukas Sekanina. "EvoApprox8b: Library of Approximate Adders and Multipliers for Circuit Design and Benchmarking of Approximation Methods." In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, pp. 258–261, 2017.

[Mig17] Szymon Migacz. "8-bit Inference with TensorRT." 2017.

[MKL18] Wojciech Mula, Nathan Kurz, and Daniel Lemire. "Faster Population Counts Using AVX2 Instructions." *Computer Journal*, **61**(1):111–120, 2018.

[MKM17] Naveen Mellempudi, Abhisek Kundu, Dheevatsa Mudigere, Dipankar Das, Bharat Kaul, and Pradeep Dubey. "Ternary Neural Networks with Fine-Grained Quantization." 2017.

[MTK17] Bradley McDanel, Surat Teerapittayanon, and H. T. Kung. "Embedded Binarized Neural Networks." pp. 1–6, 2017.

[NPH17] F. Neugebauer, I. Polian, and J. P. Hayes. "Building a Better Random Number Generator for Stochastic Computing." In *DSD*, pp. 1–8, 2017.

[NVF21] NVIDIA, Péter Vingelmann, and Frank H.P. Fitzek. "CUDA, release: 11.4.", 2021.

[NVI21] NVIDIA. "Programming Guide :: CUDA Toolkit Documentation.", 2021.

[NWC11] Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, and Andrew Y. Ng. "Reading Digits in Natural Images with Unsupervised Feature Learning." In *NIPS Workshop on Deep Learning and Unsupervised Feature Learning 2011*, 2011.

[Ope23] OpenAI. "GPT-4 Technical Report.", 2023.

[PA] Dylan Patel and Afzal Ahmad. "The Inference Cost Of Search Disruption – Large Language Model Cost Analysis.".

[PGC17] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. "Automatic differentiation in PyTorch." In *NIPS-W*, 2017.

[POZ18] Yu Pan, Peng Ouyang, Yinglin Zhao, Wang Kang, Shouyi Yin, Youguang Zhang, Weisheng Zhao, and Shaojun Wei. "A Multilevel Cell STT-MRAM-Based Computing In-Memory Accelerator for Binary Convolutional Neural Network." *IEEE Transactions on Magnetics*, **54**(11):1–5, 2018.

[PRM17] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W. Keckler, and William J. Dally. "SCNN: An Accelerator for Compressed-sparse Convolutional Neural Networks." 2017.

*[This concentrates on convo layers, where you can do re-use of packed sparse inputs. 4.3 addresses FC layers.]*

[PTT18] Fabrizio Pedersoli, George Tzanetakis, and Andrea Tagliasacchi. "Espresso: Efficient Forward Propagation for BCNNs." pp. 1–10, 2018.

[RBL21] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. "High-Resolution Image Synthesis with Latent Diffusion Models.", 2021.

[RDS15] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. "ImageNet Large Scale Visual Recognition Challenge." *International Journal of Computer Vision (IJCV)*, **115**(3):211–252, 2015.

[RHW86] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. "Learning representations by back-propagating errors." *Nature*, **323**:533–536, 1986.

[RLD17] Ao Ren, Zhe Li, Caiwen Ding, Qinru Qiu, Yanzhi Wang, Ji Li, Xuehai Qian, and Bo Yuan. "SC-DCNN : Highly-Scalable Deep Convolutional Neural Network using Stochastic Computing." In *ASPLOS*, pp. 405–418, 2017.

[RLG19] Wojciech Romaszkan, Tianmu Li, and Puneet Gupta. "3PXNet: Pruned-Permuted-Packed XNOR Networks for Edge Machine Learning." *ACM Transactions on Embedded Computing Systems (TECS)*, November 2019.

[RLG22] Wojciech Romaszkan, Tianmu Li, and Puneet Gupta. "SASCHA—Sparsity-Aware Stochastic Computing Hardware Architecture for Neural Network Acceleration." *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, **41**(11):4169–4180, November 2022.

[RLL17] Ao Ren, Ji Li, Zhe Li, Caiwen Ding, Xuehai Qian, Qinru Qiu, Bo Yuan, and Yanzhi Wang. "SC-DCNN: Highly-Scalable Deep Convolutional Neural Network using Stochastic Computing." *ACM SIGPLAN Notices*, **52**(4):405–418, 2017. arXiv: 1611.05939 ISBN: 9781450344654.

[RLM20] 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)*, pp. 768–773, 2020.

[RM87] David E. Rumelhart and James L. McClelland. *Learning Internal Representations by Error Propagation*, pp. 318–362. 1987.

[ROR16] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. "XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks." *CoRR*, **abs/1603.05279**, 2016.

[SB17] Aaron Stillmaker and Bevan Baas. "Scaling equations for the accurate prediction of CMOS device performance from 180 nm to 7 nm." *Integration, the VLSI Journal*, **58**(January):74–81, 2017.

[SB20] G. Sreegul and T. S. Bindiya. "An approximation algorithm for reducing the number of non-zero bits in the filter coefficients." In *SCEECS 2020*, pp. 1–6, 2020.

[SCS17] Ranko Sredojevic, Shaoyi Cheng, Lazar Supic, Rawan Naous, and Vladimir Stojanovic. "Structured Deep Neural Network Pruning via Matrix Pivoting." pp. 1–16, 2017.

[SFN19] Aseem Sayal, Shirin Fathima, S S Teja Nibhanupudi, and Jaydeep P Kulkarni. "All-Digital Time-Domain CNN Engine Using Bidirectional Memory Delay Lines for Energy-Efficient Edge Computing." *ISSCC*, **49**(4):228–230, 2019.

[SL17] Hyeonuk Sim and Jongeun 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.

[SNL17] Hyeonuk Sim, Dong Nguyen, Jongeun Lee, and Kiyoung Choi. "Scalable Stochastic-Computing Accelerator for Convolutional Neural Networks." In *2017 22nd ASP-DAC*, pp. 696–701. IEEE, 2017.

[SNT18] Shimpei Sato, Hiroki Nakahara, and Shinya Takamaeda-yamazaki. "BRein Memory : A Single-Chip Binary / Ternary Reconfigurable in-Memory Deep Neural Network." **53**(4):983–994, 2018.

*[Gated ternary.]*

[STM18] STMicroelectronics. "STM32 Nucleo-144 boards.", 2018.

[SZ14] Karen Simonyan and Andrew Zisserman. "Very Deep Convolutional Networks for Large-Scale Image Recognition." *arXiv preprint arXiv:1409.1556*, 2014. arXiv: 1409.1556 ISBN: 0950-5849.

[UFG17] Yaman Umuroglu, Nicholas J Fraser, Giulio Gambardella, and Michaela Blott. "FINN: A Framework for Fast, Scalable Binarized Neural Network Inference." (February), 2017.

[VAR11] R. Venkatesan, A. Agarwal, K. Roy, and A. Raghunathan. "MACACO: Modeling and analysis of circuits for approximate computing." In *ICCAD 2011*, pp. 667–673, 2011.

[VRR14] Swagath Venkataramani, Ashish Ranjan, Kaushik Roy, and Anand Raghunathan. "AxNN: Energy-efficient neuromorphic systems using approximate computing." In *ISLPED 2014*, pp. 27–32, 2014.

[VS14] Z. Vasicek and L. Sekanina. "Evolutionary design of approximate multipliers under different error metrics." In *DDECS 2014*, pp. 135–140, 2014.

[VSM11] Vincent Vanhoucke, Andrew Senior, and Mark Z. Mao. "Improving the speed of neural networks on CPUs." In *Deep Learning and Unsupervised Feature Learning Workshop, NIPS 2011*, 2011.

[VSP17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. "Attention Is All You Need." *CoRR*, **abs/1706.03762**, 2017.

[War18] Pete Warden. "Speech Commands: A Dataset for Limited-Vocabulary Speech Recognition." *CoRR*, **abs/1804.03209**, 2018.

[Wat67] Geoffrey S. Watson. "Linear Least Squares Regression." *The Annals of Mathematical Statistics*, **38**(6):1679 – 1699, 1967.

[WBS04] Zhou Wang, A.C. Bovik, H.R. Sheikh, and E.P. Simoncelli. "Image quality assessment: from error visibility to structural similarity." *IEEE Transactions on Image Processing*, **13**(4):600–612, 2004.

[WLY20] Di Wu, Jingjie Li, Ruokai Yin, Hsuan Hsiao, Younghyun Kim, and Joshua San 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.

[WZW18] Huan Wang, Qiming Zhang, Yuehai Wang, and Roland Hu. "Structured Deep Neural Network Pruning by Varying Regularization Parameters." 2018.

[YFB17] Haojin Yang, Martin Fritzsche, Christian Bartz, and Christoph Meinel. "BMXNet: An Open-Source Binary Neural Network Implementation Based on MXNet." 2017.

[YHF18] Li Yang, Zhezhi He, and Deliang 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.

[YJH21] Shimeng Yu, Hongwu Jiang, Shanshi Huang, Xiaochen Peng, and Anni Lu. "Compute-in-Memory Chips for Deep Learning: Recent Trends and Prospects." *IEEE Circuits and Systems Magazine*, **21**(3):31–56, 2021.

[YKL17] Joonsang Yu, Kyounghoon Kim, Jongeun Lee, and Kiyoung Choi. "Accurate and Efficient Stochastic Computing Hardware for Convolutional Neural Networks." *ICCD*, pp. 105–112, 2017.

[YLP17] Jiecao Yu, Andrew Lukefahr, David Palframan, Ganesh Dasika, Reetuparna Das, and Scott Mahlke. "Scalpel: Customizing DNN Pruning to the Underlying Hardware Parallelism." *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pp. 548–560, 2017.

[YME17] Amir Yazdanbakhsh, Divya Mahajan, Hadi Esmaeilzadeh, and Pejman Lotfi-Kamran. "AxBench: A Multiplatform Benchmark Suite for Approximate Computing." *IEEE Design & Test*, **34**(2):60–68, 2017.

[YSN18] Haruyoshi Yonekawa, Shimpei Sato, and Hiroki 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.

[ZDZ16] Shijin Zhang, Zidong Du, Lei Zhang, Huiying Lan, Shaoli Liu, Ling Li, Qi Guo, Tianshi Chen, and Yunji Chen. "Cambricon-X: An accelerator for sparse neural networks." *Proceedings of the Annual International Symposium on Microarchitecture, MICRO*, **2016-Decem**, 2016.

[ZHM16] Chenzhuo Zhu, Song Han, Huizi Mao, and William J. Dally. "Trained Ternary Quantization." pp. 1–10, 2016.

[ZLS18] Aidyn Zhakatayev, Sugil Lee, Hyeonuk Sim, and Jongeun Lee. "Sign-Magnitude SC : Getting 10X Accuracy for Free in Stochastic Computing for Deep Neural Networks." *DAC*, pp. 1–6, 2018.

[ZSZ17] Ritchie Zhao, Weinan Song, Wentao Zhang, Tianwei Xing, Jeng-Hau Lin, Mani Srivastava, Rajesh Gupta, and Zhiru 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.

[ZWN16] Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. "DoReFa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients." **1**(1):1–13, 2016.

[ZWT15] Qian Zhang, Ting Wang, Ye Tian, Feng Yuan, and Qiang Xu. "ApproxANN: An approximate computing framework for artificial neural network." In *DATE 2015*, pp. 701–706, 2015.