EXPLORING AVENUES TO EFFICIENTLY TRAINED DEEP NEURAL NETWORKS

by

Suryanarayanan Sathiyanarayanan

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computing

School of Computing

The University of Utah

August 2022

Copyright © Suryanarayanan Sathiyanarayanan 2022
All Rights Reserved

The University of Utah Graduate School

STATEMENT OF DISSERTATION APPROVAL

The dissertation of **Suryanarayanan Sathiyanarayanan**
has been approved by the following supervisory committee members:

Rajeev Balasubramonian ,	Chair(s)	**7/23/2022** Date Approved
Mahdi Nazm Bojnordi ,	Member	**7/20/2022** Date Approved
Liqun Cheng ,	Member	**7/18/2022** Date Approved
Mary W Hall ,	Member	**7/20/2022** Date Approved
Vivek Srikumar ,	Member	**7/20/2022** Date Approved

by **Mary W Hall** , Chair/Dean of
the Department/College/School of **Computing**
and by **David B Keida** , Dean of The Graduate School.

ABSTRACT

Over the previous decade, deep neural networks (DNNs) have become so powerful and versatile that they are applied across a variety of domains ranging from autonomous driving, web search, to drug discovery. With such broad usage it is imperative to develop hardware and software techniques to make DNNs energy efficient, and thus sustainable. Recent studies have shown that CO_2 emissions from training modern deep neural networks is non-trivial. This demands an emphasis on energy efficiency not only for inference, but also for training.

We explore two contrasting ideas both aimed towards reducing energy for training. First, we conduct an exhaustive design space exploration of different partitioning and pipelining strategies used to distribute DNN training. With insights gained from the design space exploration, we propose Cafine - a criticality-aware fine-grained pipelining technique. Cafine first profiles and groups DNN layers, and then classifies layer-groups based on their criticality to throughput. It then applies temporal pipelining to critical layer-groups and spatial pipelining to non-critical layer-groups to reduce energy consumption. Though Cafine makes training more efficient, the improvement is only incremental. In order to achieve significant strides, training needs something transformational and has to be approached from a different angle.

We hypothesize that using spiking neural networks (SNNs) can improve training efficiency. SNNs are built with spiking neurons as opposed to digital neurons used in conventional artificial neural networks (ANNs). SNNs support STDP (Spike Time Dependent Plasticity), an online unsupervised training technique. SNNs also encode data as 1-bit spikes and process inputs with addition operations, which makes them less complex than ANNs. These low-energy features combined with the compatibility to online unsupervised learning algorithms makes SNNs a viable choice for designing energy efficient DNNs. Before we can reap these benefits, SNNs have two main challenges that needs to be addressed – low efficiency accelerators and low accuracy. Existing commercial SNN

accelerators have throughput/efficiency that is 2 orders of magnitude lower than commercial ANN accelerators. Moreover accuracy achieved by STDP-trained SNNs are behind that of ANNs trained using stochastic gradient descent. Addressing these challenges is vital so we can move towards DNNs trained in an efficient and unsupervised manner. In this thesis, we tackle the challenge of efficient SNN acceleration. We first propose INXS, an analog accelerator that provides orders of magnitude energy efficiency improvement compared to the IBM TrueNorth. Analog accelerators however, continue to be a challenge for real-world deployment. So we next investigate dataflows and architectures for a digital SNN accelerator. To better understand the nature of SNNs and ANNs, we conduct an elaborate study across diverse network topologies, sparsity ratio, and resolution. Based on the observations, we propose SpinalFlow, a digital accelerator that establishes the potential of SNNs and scenarios where SNNs can achieve better efficiency than ANNs.

For my parents, Sathiya and Vasu.

CONTENTS

ABSTRACT . . . . . **iii**

LIST OF FIGURES . . . . . **ix**

LIST OF TABLES . . . . . **xii**

ACKNOWLEDGEMENTS . . . . . **xiii**

CHAPTERS

1. INTRODUCTION **1**

1.1 Dissertation Overview 3
1.1.1 Dissertation Statement 3
1.1.2 Criticality-Aware Pipeline-Parallel Distributed Training 3
1.1.3 In-situ Analog Accelerator for Spiking Neural Networks 4
1.1.4 Digital Accelerator for Spiking Neural Networks 4

2. UNDERSTANDING THE IMPACT OF PIPELINED EXECUTION STRATEGIES FOR DNN TRAINING **6**

2.1 Introduction 6
2.2 Background 8
2.2.1 Primer on Neural Network Training 8
2.2.2 Distributed Training 9
2.2.2.1 Data and Model Parallel Training 9
2.2.2.2 Pipeline Parallel Training 9
2.2.3 Hardware for Training 9
2.3 Exploring Different Execution Strategies 10
2.3.1 Methodology 11
2.3.1.1 System Setup 11
2.3.1.2 Execution Strategy 11
2.3.1.3 Analytical Model 12
2.3.2 Design Space Exploration 13
2.3.2.1 Convolutional Networks 13
2.3.2.2 Fully Connected Networks 15
2.4 Cafine 16
2.4.1 Chip-Level Execution Strategies 16
2.4.2 Cafine: Criticality-Aware Fine-Grained Pipeline 17
2.4.3 Results 18
2.4.3.1 Convolutional Networks 18
2.4.3.2 Fully Connected Networks 19
2.4.3.3 Discussion for All Workloads 19
2.4.3.4 Discussion on Memory Requirements 20

2.5 Related Work 20

3. INXS: BRIDGING THE THROUGHPUT AND ENERGY GAP FOR SPIKING NEURAL NETWORKS 29

3.1 Introduction 29
3.2 Background 31
3.2.1 Spiking Neurons 31
3.2.2 SNN Accelerators 32
3.2.3 ANN Accelerators 33
3.3 The INXS Architecture 35
3.3.1 Overview 35
3.3.2 Implementation Details 36
3.3.2.1 Overall Chip Organization 36
3.3.2.2 The Odd Phase 37
3.3.2.3 The Even Phase 37
3.3.2.4 An Example Design Point 38
3.3.2.5 Balancing the Pipeline 39
3.3.2.6 Neuron Model 40
3.3.2.7 Routing Table 40
3.4 Methodology 40
3.5 Results 42
3.5.1 INXS Design Space Exploration 42
3.5.2 Comparison to TrueNorth 43

4. SPINALFLOW: AN ARCHITECTURE AND DATAFLOW TAILORED FOR SPIKING NEURAL NETWORKS 51

4.1 Introduction 51
4.2 Background 53
4.2.1 Spiking Neurons 53
4.2.2 SNN Accelerators 55
4.2.3 ANN Accelerators 55
4.3 Understanding Sources of SNN Inefficiency 56
4.3.1 Defining the ANN and SNN Baselines 56
4.3.1.1 ANN Baseline 56
4.3.1.2 SNN Baseline 56
4.3.2 Evaluation Parameters 57
4.3.3 Analysis of Spiking Eyeriss 58
4.3.4 Summary 59
4.4 Proposed SNN Architecture: SpinalFlow 59
4.4.1 Terminology 59
4.4.2 Hardware Organization 60
4.4.3 Summary 62
4.4.4 Hardware Details 62
4.4.5 Other Networks 63
4.5 Results 64
4.5.1 SpinalFlow versus Eyeriss 64
4.5.1.1 Energy Comparison 64

4.5.1.2 Effect of Low Resolution 65
4.5.1.3 Latency Comparison 65
4.5.2 SpinalFlow versus Spiking-Eyeriss 66
4.5.3 Fully-Connected Layers 67
4.5.4 Scalability Study 67
4.5.5 SpinalFlow versus SCNN 67
4.6 Re-Visiting the SNN versus ANN Debate 68
4.6.1 Comparing SNN versus ANN Efficiency 68
4.6.2 Discussion of Prediction Accuracy 69

5. CONCLUSIONS AND FUTURE WORK 84

5.1 Criticality-Aware Pipeline-Parallel Training 84
5.2 Re-RAM Based Analog In-situ SNN Accelerator 85
5.3 Digital SNN Accelerator and Dataflow 85
5.4 Support for STDP-Based Training in INXS and SpinalFlow 86
5.4.1 STDP-Based Training 86
5.4.2 Hardware Support for STDP in SpinalFlow 86
5.5 Concluding Remarks 87
5.6 Future Work 87

REFERENCES 90

LIST OF FIGURES

2.1 Chip (a), PE tile (b), and buffer tile (c) architecture. 22

2.2 Illustration of different components of execution strategy for an example DNN, cluster, chip, and layer. 23

2.3 Brief overview of the performance model used in the analytical simulator. 23

2.4 Exploring the design space of ResNet-50 training by varying number of groups, number of chips, batch size, and partitioning strategy. xC refers to a design with x chips and xG refers to a config where ResNet-50 is partitioned into x groups. 23

2.5 Throughput of data parallel training of ResNet-50. 24

2.6 Energy efficiency of ResNet-50 for different configurations. 24

2.7 Exploring the design space of Bert-base training by varying number of groups, number of chips, batch size, and partitioning strategy. xC refers to a design with x chips and xG refers to a config where Bert-base is partitioned into x groups. 24

2.8 Size of activation/error and weight tensors of Bert-base at different batch sizes. 25

2.9 Efficiency of training Bert-base for different configurations. 25

2.10 Bert-base throughput for data/model parallelism. 25

2.11 Execution latency of all groups in ResNet-50 for PipeDream and FG-Pipe. Red circle denotes the execution strategy chosen for the specific group by Cafine. 26

2.12 Group-wise energy consumption of ResNet-50 for PipeDream and FG-Pipe. Red circle denotes the execution strategy chosen for the specific group by Cafine. 26

2.13 Chip level execution strategies for ResNet-50. 27

2.14 Chip level execution strategies for Bert-base. 27

2.15 Comparing with PipeDream. (a) Performance and (b) Energy of different execution strategies relative to PipeDream. **CNN-A**: Average across CNN workloads (**ResNet-34**, **ResNet-50**, **VGG-D**), **FC-A**: average across FC workloads (**Bert-base**, **Bert-large**, **GPT2-XL**), **Avg**: average across all the six workloads. . . 28

2.16 Performance and memory requirements of Cafine compared to a fully temporal pipeline. 28

3.1 A basic 2-input LLIF spiking neuron. The figure shows how the neuron potential is incremented when input spikes are received, how a leak is subtracted when there are no input spikes, and how an output spike is produced when the potential crosses the threshold. 45

3.2 Example of an input image being converted into a number of input spike trains that are fed to a spiking neural network. 45

3.3 In-situ computation using memristor xbar. (a) An example 4x4 memristor xbar connected to peripheral circuits. (b) The conductance of the memristor corresponds to the synaptic weight and the voltage corresponds to the spike input. The current at the end of the bitline is the dot-product of input spikes and synaptic weights. 46

3.4 INXS tiled architecture and details of one tile. 46

3.5 INXS architecture. (a) Synaptic Unit. (b) Neuron Unit. (The exact number of xbars, ADCs, adders, etc. vary in our optimal design points.) 47

3.6 INXS pipeline. IB - Input Buffer, CB - Central Buffer, OB - Output Buffer 47

3.7 Computational efficiency of INXS for various configurations. 47

3.8 Energy efficiency of INXS for various configurations. 48

3.9 Storage efficiency of INXS for various configurations. 48

3.10 M-synapses/mm^2 and K-Neurons/mm^2 for various INXS configurations. 48

3.11 Energy (for 1 image) estimates of different INXS configurations for VGG-NET. 49

3.12 Energy (for 1 image) estimates of different INXS configurations for MSRA. . . . 49

3.13 Energy estimate of INXS (8x8x8x128) for different layers of VGG-NET. 49

3.14 Energy estimate of INXS (8x8x8x128) for different layers of MSRA. 50

4.1 A basic 2-input LLIF spiking neuron. The figure shows how the neuron potential is incremented when input spikes are received, how a leak is subtracted when there are no input spikes, and how an output spike is produced when the potential crosses the threshold. 73

4.2 Example of an input image converted into a number of input spike trains that are fed to a rate-coded SNN (r-SNN). 74

4.3 Architectures Evaluated. (a) PE in Eyeriss [14]. (b) PE in Spiking-Eyeriss. 74

4.4 Energy consumed per inference by r-SNN on Spiking-Eyeriss normalized to Eyeriss. Sp60, Sp90, and Sp98 refers to 60%, 90%, and 98% sparsity respectively. 75

4.5 Energy consumed per inference by t-SNN on Spiking-Eyeriss normalized to Eyeriss. 75

4.6 A spine in a CONV layer. 76

4.7 Example: Step1, Cycle 1. 76

4.8 Example: Step1, Cycle 2. Step 1 continues until all receptive field entries (up to 2K) have been processed. 77

4.9 Example: End of Step 1 and set-up before Step 2. 77
4.10 SpinalFlow. (a) Chip. (b) PE details. (c) Dataflow pseudocode. 78
4.11 Energy/inference of SpinalFlow normalized to an 8-bit Eyeriss with 60% sparsity. Sp60, Sp90, and Sp98 refers to 60%, 90% and 98% sparsity for the SNN. 78
4.12 Energy/inference of ResNet at 8b resolution and 60% sparsity. (a) On Eyeriss, (b) On SpinalFlow. ifmap, filt and psum refers to corresponding scratch-pads in Eyeriss PE. 78
4.13 Energy per inference for SpinalFlow for full workloads, normalized to 8bSp60 Eyeriss. 79
4.14 Energy/inference of SpinalFlow, normalized to an Eyeriss baseline with the same resolution as SpinalFlow. Note that sparsity of Eyeriss is fixed at 60%. . . 79
4.15 Energy/inference of SpinalFlow for full workloads, normalized to an Eyeriss baseline with the same resolution as SpinalFlow. 80
4.16 Latency per inference of SpinalFlow with respect to Eyeriss (a) with 72 GLB links and (b) with 1024 GLB links. 80
4.17 SpinalFlow energy per inference normalized to Spiking-Eyeriss for synthetic conv workloads. 81
4.18 SpinalFlow energy per inference normalized to Spiking-Eyeriss for full network workloads 82
4.19 Energy per Inference of the synthetic fully connected workloads for SpinalFlow normalized to Eyeriss. 82
4.20 Energy per inference for ResNet on SpinalFlow, normalized to Eyeriss, as a function of the number of PEs in SpinalFlow. 83
4.21 Compute density ($GOPS/mm^2$) of SpinalFlow normalized to Eyeriss and its area as the number of PEs is varied. 83

LIST OF TABLES

2.1 Chip specifications. 21

2.2 Optimal execution strategies for CNN and FC workloads based on batch size and priority metric – performance, energy efficiency, EDP. DP/MP are data/model parallelism. 21

3.1 Comparison of accuracy, throughput, and energy efficiency for state-of-the-art ANNs and SNNs. The digital SNN numbers correspond to TrueNorth [4]. The energy number for the Analog SNN accelerator is for a small-scale 32-neuron implementation [78]. 44

3.2 INXS area and power breakdown (for one 8x8x8x128 tile configuration). SU: Synaptic Unit, NU: Neuron Unit. 44

3.3 INXS comparison with TrueNorth. ND-Neuron density, SD-Synaptic density . 45

4.1 Parameters for our ANN (based largely on Eyeriss) and baseline SNN. 71

4.2 Workloads, degree of sparsity, and resolution. SC - Standard Conv, DWC - Depth-Wise separable Conv, PWC - Point-Wise separable Conv. The SNN network from [63] will be referred to as STDP-Net for the rest of the chapter. . . 71

4.3 Architecture specifications of SpinalFlow and Eyeriss-1K. FB- Filter Buffer, GLB - Global Buffer, B/W - Bandwidth, A - Area in mm^2, P - Power in mW . . . 72

4.4 Accuracy comparison with supervised training on labeled datasets. 73

5.1 Area (in mm^2) of SpinalFlow without and with support for STDP-based training. 89

ACKNOWLEDGEMENTS

First I would like to thank my mother (Vasumathi), my father (Sathiyanarayanan), my wife (Elavarasi), my under-grad advisor (Prof. Waran), and my Ph.D. advisor (Prof. Balasubramonian), who are as much a part of this Ph.D. as me. Their roles have been crucial in both starting as well as reaching the finish line. I will always be grateful for their support.

I thank my committee members, Mahdi, Vivek, Mary, and Liqun, for their valuable feedback on my dissertation. I would like to thank my friends and labmates, Meysam, Anirban, Sumanth, Karl, Ali, Sarabjeet, and Chandru for playing a big part in helping me navigate through my Ph.D. I would also like to thank the School of Computing staff, Karen, Ann, Robert, Chethika, and Lauren, for their incredible support from the administrative side.

CHAPTER 1

INTRODUCTION

A fundamental concept driving the success of machine learning is deep neural networks. With the inception of AlexNet [68], DNNs have burst into the machine learning scene, and are the pivotal technique used in applications spanning image classification, object detection, translation, language modelling, etc. Due to its wide-spread use, DNNs are being deployed across a plethora of platforms, with diverse performance, power, and area constraints. A few examples include, Nvidia NVDLA [114] used in IoT devices, Tesla FSD [126] used in automobiles, and Google TPU [96] used in data centers.

With such wide-spread use, DNN models are trained/updated at a very high frequency. For example, over a period of 18 months Facebook saw a 7× increase in jobs submitted for distributed training in their data centers [93]. DNNs are trained offline, typically in a data center over millions of samples. Depending on the model size and resources available, training a DNN could take from a few hours to weeks/months. A 2019 report estimated that training an NLP (Natural Language Processing) model using neural architecture search [144] emits more than 300× the CO_2 emitted by a flight between New York City and San Francisco [119]. Thus, improving training efficiency is more important than ever.

Prior works have taken varied paths to improve training efficiency. This include techniques like quantization [86, 130], specialized systems/architectures [12, 79, 128], compression/sparsity [11, 52, 76], hybrid partitioning [53, 54, 67, 117, 118], pipeline-parallel training [43, 48, 136], etc. In this thesis, we first explore heterogeneous partitioning/pipelining strategies. Within a device, a DNN can be executed in two ways: 1. *temporal pipelining*, where layers are executed one after the other with each layer using all the available resources, and 2. *spatial pipelining*, where layers in a DNN share the available resources and execute in a pipelined fashion. Also, there are many ways to partition lay-

ers within and across devices. There isn't a clear understanding of how these different partitioning and pipelining techniques jointly affect training performance. We first conduct a thorough analysis of different pipelining/partitioning strategies on different DNNs, batch sizes, and cluster sizes. Building upon the observations of the analysis, we propose Cafine, a criticality-aware fine-grained pipelining technique. Cafine first profiles layers of a DNN and groups them such that workload across groups is balanced to facilitate efficient pipeline-parallel training. Next, if a layer-group is critical to training throughput, Cafine schedules it to be executed in a temporal pipeline. If a layer-group is not critical, it gets scheduled to be executed in a spatial pipeline. Cafine also determines the optimal dimension(s) across which each layer has to be partitioned. Thus, by choosing the optimal partitioning/pipelining strategy, Cafine reduces DNN training energy without compromising performance/accuracy.

Though techniques to improve traditional DNN training methods are necessary, the improvements they provide have either plateaued or have become incremental. We next explore something that has the potential to provide dramatic advancements – SNNs. SNNs can be trained with STDP (Spike Time Dependent Plasticity), a biologically plausible unsupervised online learning process [34]. This makes SNNs a good choice especially in cases where labelled data is hard to get and where continual learning is required. Furthermore, SNNs with Linear Leaky Integrate and Fire neurons (LLIF) can be a cheaper alternative to traditional neural networks. This is due to two key features: 1. representing information as 1-bit spikes, and 2. processing non-zero inputs with additions (instead of multiplications). But currently SNNs suffer from two major obstacles: 1. less efficient and low throughput accelerators, and 2. low accuracy in complex ML tasks. In this thesis we focus on the first challenge – improving the performance and efficiency of SNN accelerators, by introducing novel dataflow and architecture.

There have been several works aimed at accelerating SNNs [4, 28, 62], however, their flexibility, performance and efficiency have been limited. There also exists a gap in understanding of the potential and efficiency offered by SNNs. We conduct a comprehensive comparison of SNNs and ANNs across a variety of network architectures, dataflows and activation sparsity. We also design both digital and analog SNN accelerators, that improve SNN energy efficiency significantly compared to IBM TrueNorth. Finally, we quantify

scenarios (resolution, sparsity, network topology) where SNNs prove to be an efficient alternative to ANNs.

1.1 Dissertation Overview

1.1.1 Dissertation Statement

We hypothesize that the training efficiency of deep neural networks can be improved by (1) heterogeneous execution strategies to distribute training, (2) creating competitive implementations for spiking neural networks that provide high sparsity at low resolution and a more localized energy-efficient training algorithm . We test this hypothesis in the context of a criticality-aware pipeline-parallel training technique, an in-situ analog SNN accelerator, and a digital SNN accelerator.

1.1.2 Criticality-Aware Pipeline-Parallel Distributed Training

Deep neural networks (DNNs) are typically trained on clusters of GPUs/TPUs for days, while consuming significant amounts of energy. In order to distribute training across a cluster of worker nodes, there are many choices regarding how the workload is pipelined and partitioned. Each of these choices has a direct impact on both the performance and energy efficiency of training. We refer to this set of choices as the *execution strategy*. While prior works have introduced pipeline-parallel training, we explore a broad design space to better understand the deployments where they are effective. In particular, we observe that pipeline-parallel training is effective for image models at small batch sizes and small group counts; whereas in language models, it is effective only at large batch sizes and moderate group counts. Thus, the workload's neuron-to-weight ratio plays a significant role in determining if temporal or spatial pipelines are better and if model or data parallelism is better.

Based on the insights gained from the design space exploration, we introduce Cafine, a **C**riticality-**a**ware **fine**-grained pipeline that achieves better energy efficiency than state-of-the-art pipeline-parallel training techniques without loss in performance. In pipeline-parallel training, different groups have different execution times because of the variation in work and resources assigned to each group. The slowest group is on the critical path and determines overall throughput. Cafine employs different execution strategies for critical and non-critical pipeline stages, such that energy is reduced by 6% on average, and by a maximum of 11%, compared to a state-of-the-art PipeDream baseline, while having the

same throughput. We also note that the optimal execution strategy depends on the target DNN workload and the power/performance requirements.

1.1.3 In-situ Analog Accelerator for Spiking Neural Networks

In recent years, multiple neuromorphic architectures have been designed to execute cognitive applications that deal with image and speech analysis. These architectures have followed one of two approaches. One class of architectures is based on machine learning with artificial neural networks. A second class is focused on emulating biology with spiking neuron models, in an attempt to eventually approach the brain's accuracy and energy efficiency. A prominent example of the second class is IBM's TrueNorth processor that can execute large spiking networks on a low-power tiled architecture, and achieve high accuracy on a variety of tasks. However, as we show in this work, there are many inefficiencies in the TrueNorth design. We propose a new architecture, INXS, for spiking neural networks that improves upon the computational efficiency and energy efficiency of the TrueNorth design by 3,129$\times$ and 10$\times$ respectively. The architecture uses memristor crossbars to compute the effects of input spikes on several neurons in parallel. Digital units are then used to update neuron state. We show that the parallelism offered by crossbars is critical in achieving high throughput and energy efficiency.

1.1.4 Digital Accelerator for Spiking Neural Networks

Spiking neural networks (SNNs) are expected to be part of the future AI portfolio, with heavy investment from industry and government, e.g., IBM TrueNorth, Intel Loihi, Qualcomm Zeroth. While ANN architectures have taken large strides, few works have targeted SNN hardware efficiency. Our analysis of SNN baselines shows that at modest spike rates, SNN implementations exhibit significantly lower efficiency than accelerators for ANNs. This is primarily because SNN dataflows must consider neuron potentials for several ticks, introducing a new data structure and a new dimension to the reuse pattern. We introduce a novel SNN architecture, SpinalFlow, that processes a compressed, time-stamped, sorted sequence of input spikes. It adopts an ordering of computations such that the outputs of a network layer are also compressed, time-stamped, and sorted. All relevant computations for a neuron are performed in consecutive steps to eliminate neuron potential storage overheads. Thus, with better data reuse, we advance the energy

efficiency of SNN accelerators by an order of magnitude. Even though the temporal aspect in SNNs prevents the exploitation of some reuse patterns that are more easily exploited in ANNs, at 4-bit input resolution and 90% input sparsity, SpinalFlow reduces average energy by 1.8×, compared to a 4-bit Eyeriss baseline. These improvements are seen for a range of networks and sparsity/resolution levels; SpinalFlow consumes 5× less energy and 5.4× less time than an 8-bit version of Eyeriss. We thus show that, depending on the level of observed sparsity, SNN architectures can be competitive with ANN architectures in terms of latency and energy for inference, thus lowering the barrier for practical deployment in scenarios demanding real-time learning.

CHAPTER 2

UNDERSTANDING THE IMPACT OF PIPELINED EXECUTION STRATEGIES FOR DNN TRAINING

2.1 Introduction

Deep neural network (DNN) training is a resource-intensive task, given the large size of models, the many hyper-parameters that are explored, the large training data-sets, and the many epochs required for high accuracy. Recent reports [99, 108, 119, 132] have quantified the high environmental impact of DNN training, underlining the urgency to develop algorithmic, systems, and hardware innovations to reduce its energy footprint. Training a large model with a large dataset typically requires a dedicated cluster of compute nodes for several days [115, 121]. The compute nodes can be CPUs, GPUs, TPUs, or other accelerators [1, 2, 29, 35, 56, 135].

In order to distribute DNN training across a cluster of nodes, there are several options in terms of work partitions and pipelined execution, which impact performance and energy. Execution of a DNN can be pipelined in the following broad ways: 1. *Temporal Pipelining*, where all nodes work on the same layer in parallel and execution of different layers is pipelined over time, and 2. *Spatial Pipelining*, where the DNN is partitioned into multiple groups and each group is assigned a different set of nodes. AccPar [118] is an example of a temporal pipeline, and PipeDream [43] is an example of a spatial pipeline. Once the pipelining strategy is fixed and layers are assigned a set of nodes, the next choice is the optimal partition of layers across those nodes. Layers can be partitioned along the batch dimension, input dimension, output dimension, etc. Two common approaches are: *Data parallelism*, where each worker handles a subset of the data for all layers, and *model parallelism*, where each worker handles a subset of the model while working on the entire dataset. Works like One Weird Trick [67], HyPar [117], and AccPar [118] have explored the effectiveness of different partitioning approaches. We refer to the combined pipelining

and partitioning approach as the *execution strategy*.

While prior works have studied pipelining and partitioning strategies [25, 33, 43, 53, 59, 60, 67, 81, 117, 118, 136], they have primarily focused either only on pipelining (e.g., PipeDream) or partitioning (e.g., AccPar) but not both. AccPar considers different partitioning strategies for different layers, but the execution of layers always uses a temporal pipeline. PipeDream breaks up the DNN into layer groups; each layer group is assigned a different set of nodes, thus forming a spatial pipeline. Within a layer group, one layer executes at a time and the overall work is partitioned across nodes along the batch dimension, i.e., data parallelism with a temporal pipeline. So far, pipelining and partitioning approaches have been explored in a disjoint manner; we help fill a gap in the analysis literature by exploring a broad design space that considers both, as well as the impact of batch size, group size, cluster size, and workload on throughput and energy.

Through this exploration, we make two observations that deviate from current assumptions about partitioned/pipelined training. First, for convolutional networks, pipeline-parallel training (spatial pipeline) yields better performance only at small/moderate batch sizes. As batch size increases, the overhead of inter-group communication dominates, which results in fully temporal pipelines emerging as the optimal execution strategy. Second, for fully connected networks used by language models, partitioning layers along the model dimension achieves better performance than data parallelism (batch partitioning) only for small/moderate batches. With a large batch size, inter-layer communication of activations/errors is more than gradient aggregation. As a result, pipeline-parallel execution with data parallelism and a moderate number of groups emerges as the optimal strategy for large batches. Large batches not only boost throughput, but studies have shown that the batch size can be safely increased even up to 32K without any loss in accuracy [39, 138–140].

Our initial design space exploration of the PipeDream approach only uses a temporal pipeline within a layer group, but with partitions along varying dimensions (batch, channel, kernel). Note that monolithic architectures like the TPU only support temporal pipelines. However, with the advent of tiled architectures [41, 113, 128], resources can be allocated to layers at a finer granularity [38]. This opens up the possibility of implementing a spatial pipeline within a layer group. We consider a chip-level fine-grained spatial

pipeline such that a layer can start processing once a slice of activations is produced by the previous layer. By not waiting for the previous layer to produce the entire activation tensor, buffering requirements are brought down, which reduces the total memory accesses and the associated interconnect overheads. We observe that such fine-grained spatial pipelines can reduce energy while incurring a small performance penalty. We then exploit this performance-energy trade-off in a PipeDream setting by observing that layer groups in PipeDream have different execution times. The slowest layer group is on the critical path and determines overall throughput, while other groups are non-critical. We introduce *Cafine*, a **C**riticality-**a**ware **fine**-grained pipeline, that exploits this *latency slack*, and reduces energy by invoking a spatial pipeline for non-critical layer groups and a temporal pipeline for the critical layer group. Moreover, we observe that the optimal execution strategy varies not only based on the target DNN, batch size, etc., but also based on the priority metric – throughput, energy efficiency, EDP, and memory capacity.

In summary, this chapter makes the following contributions:

1. We carry out a comprehensive design space exploration of different combinations of partitioning and pipelining strategies for vision and language models to reveal their performance and energy characteristics.

2. We observe that conventional partitioning and pipelining strategies do not extend to larger batches. For large batches, ResNet-50 achieves 7.4× higher throughput with a temporal pipeline than with pipeline-parallel training. Bert-base at a large batch size performs 1.9× better with data parallelism than with model parallelism, again, contrasting from conventional wisdom.

3. Finally, we propose Cafine, which selectively applies fine-grain spatial pipelining to non-critical groups while adopting a cluster-level spatial pipeline. Cafine reduces energy by 6% on average and by a maximum of 11%, compared to PipeDream without any loss in performance.

2.2 Background

2.2.1 Primer on Neural Network Training

Neural network training is a highly iterative process requiring tens of epochs and millions of labeled inputs in each epoch [68, 71, 121]. Additionally, with techniques like

AutoML and Neural Architecture Search (NAS) that automate the design of machine learning models, training cost is further increased [100, 143, 144].

In order to bring down training time, DNNs are typically distributed across a cluster of worker nodes. The execution strategy and the worker's underlying architecture are two of the most important factors in deciding the performance and efficiency of training. Many works have tried to improve both, which we will briefly discuss next.

2.2.2 Distributed Training

2.2.2.1 Data and Model Parallel Training

In data parallelism, each worker performs the forward and backward passes for a subset of inputs in a batch, which requires the entire model to be replicated in every worker. After the backward pass, weight gradients are calculated by every worker, which then gets aggregated before performing a parameter update. The next iteration of training begins after the updated weights are broadcast to all the workers. The need to replicate weights and update them after every batch are major drawbacks of data parallelism. As large models cannot be distributed with data parallelism due to memory constraints, model parallelism assigns a subset of layers to each worker for all inputs in the batch. However, studies have shown that model parallelism can lead to idling among nodes [43].

2.2.2.2 Pipeline Parallel Training

Pipeline-parallel training overcomes the drawbacks of both data and model parallelism. In pipeline-parallelism, the DNN is partitioned into multiple stages or groups, with each group comprising of multiple consecutive layers. A group is then assigned to one or more workers depending on its compute/memory requirements. The computations for a set of inputs execute on the first set of workers; their outputs are then fed to the worker handling the next group, and so on, thus establishing a *spatial pipeline*. While we focus on the PipeDream [43] implementation, other variants also exist that vary in their weight updates, granularity of pipelining, and input parallelism [33, 43, 48, 66, 136].

2.2.3 Hardware for Training

Several projects have introduced hardware targeted at DNN training. Such chips can be broadly classified into monolithic and tiled architectures. TPU versions 1-4 [56, 96], Tesla

Dojo [2], etc. fall under the monolithic category, and architectures like ScaleDeep [128], SIMBA [113], Cerebras [10], GraphCore [41], and Intel NNP [135] are examples of tiled architectures. A monolithic architecture has one or two large systolic arrays and one or two large central buffers (similar to the Google TPUs) that can only work on a subset of one DNN layer at a time. Monolithic architectures can only execute a temporal pipeline. A tiled architecture has multiple independent tiles on a single chip, each with vector/systolic units and private buffers. This work primarily focuses on tiled architectures because they offer flexibility when mapping tasks to the underlying hardware, and the layers can be executed both in a temporal pipeline with all tiles working on a single layer before advancing to the next layer, or in a spatial pipeline, with adjacent tiles executing consecutive layers of the DNN.

2.3 Exploring Different Execution Strategies

PipeDream achieves significant speedup over data and model parallelism. However, the research literature has explored a limited design space:

1. The layers in each group were only partitioned across worker nodes along the batch dimension, i.e., data parallel training. A thorough study of how different partitioning strategies (batch, input, output, etc.) impact pipeline-parallel training was not performed.

2. Since PipeDream was proposed, the Transformer architecture [127] has been introduced, whose encoder/decoder blocks form the basis for several state-of-the-art NLP models [23, 26, 80, 102]. The impact of pipeline-parallel training on such NLP models is not well established.

3. The performance of pipeline-parallel training has been evaluated; the impact on energy has not been analyzed.

4. PipeDream only considers a single batch size; we extend the analysis to consider the larger batch sizes that have been shown to be effective in many cases [39, 138–140].

This work attempts to fill some of the gaps in this exploration. We expand the design space, ranging from temporal pipelines (e.g., AccPar) to pipeline-parallel training (e.g.,

PipeDream). Specifically, we explore different layer grouping scenarios, layer partitioning strategies, batch sizes, and cluster sizes for various DNNs.

2.3.1 Methodology

2.3.1.1 System Setup

We assume a baseline generic tiled architecture that is fine-tuned for training by borrowing ideas from different state-of-the-art designs. As weight stationary dataflow is widely adopted [56, 113, 114], we consider an NVDLA [114] inspired tile as our building block processing element, shown in Figure 2.1b. Since training requires significantly more on-chip memory than inference, we also consider a buffer tile, consisting of a large global buffer (like an L2 scratchpad) and vector units (Figure 2.1c). Such structures are also present in other architectures, including SIMBA [113] and SCALEDEEP [128]. As shown in Figure 2.1a, a single chip or node consists of alternating columns of PE tiles and buffer tiles. In total, each chip consists of 9 columns and 8 rows of buffer tiles and 8 columns and 8 rows of PE tiles, arranged in a 17×8 2D-grid. Four such chips constitute a node, and nodes are connected with PCIe interconnects. Each node is connected to 32 GB of HBM memory.

2.3.1.2 Execution Strategy

For a given DNN and a given cluster, different choices about pipelining, mapping, and partitioning can be made. We define these choices collectively as *execution strategy*.

① **Cluster-level pipelining:** A DNN can be distributed either in a cluster-level temporal pipeline (e.g., AccPar) or in a spatial pipeline (e.g., PipeDream). Figure 2.2(b) and 2.2(c) show cluster-level spatial and temporal pipelining of the example DNN shown in Figure 2.2(a).

② **Mapping:** Mapping applies only in the case of a cluster-level spatial pipeline. It refers to the physical allocation of chips to layer groups such that inter-group communication is minimized. Figure 2.2(b) shows how the four groups are mapped across 16 chips.

③ **Chip-level pipelining:** Once chips are allocated to groups, layers in each group can be executed within each chip in a temporal or a spatial pipeline. This is valid

only for tiled architectures, which is our baseline for the design space exploration. Chip-level spatial and temporal pipelining of layers 5 and 6 in group-3 are shown in Figure 2.2(d) and 2.2(e), respectively.

④ **Inter-chip partitioning:** When a group is assigned to multiple chips, each layer in the group can be partitioned along different dimensions (B,H,W,C,K in Figure 2.2) to be parallelized across the chips. Figure 2.2(f) illustrates partitioning layer-5 across 4 chips along the batch dimension, assuming it is executed in a cluster-level spatial pipeline.

⑤ **Inter-tile partitioning:** Once a portion of a layer is assigned to a chip, it has to be further partitioned across the tiles within a chip. Like inter-chip partitioning, inter-tile partitioning can also be performed along the four dimensions: B, H, W, C, K. Inter-tile partitioning applies only for tiled architectures. Figure 2.2(g) shows inter-tile partitioning of layer-5 along both channel/input and kernel/output dimensions, assuming layer-5 is executed in a chip-level temporal pipeline.

In this study, we focus on how cluster-level pipelining ① and inter-chip partitioning ④ strategies behave for varying batch sizes. For each design point, we pick the optimal mapping ② and inter-tile partitioning ⑤ strategies. Impact of chip-level pipelining strategies ③ is discussed in Section 3.3.

2.3.1.3 Analytical Model

We developed an analytical simulator that models accesses to on-chip buffers, external memory (HBM), and data transferred on inter-tile and inter-chip links.

Performance Model. These models are then used to estimate the performance and energy consumption of different design points. Figure 2.3 shows an overview of our computation and communication cost model for estimating training throughput. Due to space limitations, we keep the explanation brief and expand only on a few inter-layer communication cost models.

Energy Model. The PE-tiles and buffer-tiles were modeled in Verilog and synthesized using Synopsis Design Compiler at 28 nm FDSOI technology node. To account for the area and placement overhead of on-chip buffers, we consider them as black boxes during the

backend flow, based on the dimensions provided by CACTI [89]. Using the dimensions of PE and buffer tiles, we estimate the length of inter-tile interconnects. Inter-tile communication overhead is calculated by multiplying the energy per unit wire length obtained from CACTI [89] and the calculated inter-tile wire length. Table 2.1 summarizes the architectural specifications of our baseline architecture.

Our energy model estimates the number of accesses to all buffers, HBM, and the data transferred on across all interconnects. This is combined with the per-access energy numbers summarized in Table 2.1 to estimate system energy.

For our experiments, we consider six workloads from vision and NLP domains – ResNet-34 and ResNet-50 [45], Bert-base and Bert-large [26], VGG-D [115], GPT-2 X-large [102]. We consider the common mixed-precision training technique, where activations/errors/weights are 16-bits, accumulation is at 32-bits, and the parameter updates are carried out by a 32-bit master copy of weights [86].

2.3.2 Design Space Exploration

2.3.2.1 Convolutional Networks

Recall that *execution strategy* is the combination of the cluster-level pipeline (involving one or more groups), the chip-level pipeline, and the partitioning strategy within a group. We first examine these strategies for convolution-heavy networks like ResNet-50.

Figure 2.4 shows throughput for different group and cluster sizes at batch sizes of 128 and 8192, while considering both data and model parallelism. For now, we are only considering the impact of the cluster-level pipeline (by varying the number of groups) and the partition strategy (data vs. model parallelism). In data parallelism, a layer is partitioned such that each worker node operates on a portion of the mini-batch. Whereas in model parallelism, a layer can be partitioned along the input (channel) dimension or output (kernel) dimension. Though input and output partitioning have different communication overheads for forward and backward passes, the overall communication costs are similar. Therfore, for the remainder of the chapter, data and model parallelism refers to partitioning layers along the batch and input dimension respectively. As observed by prior works, data parallelism incurs inter-node communication of gradients whereas model parallelism requires communication of activations/errors. As convolutional layers

have significantly more activations/errors than gradients, data parallelism performs better than model parallelism for all design points.

- **Impact of grouping:** Increasing the number of groups results in better performance up to a point, as shown in Figure 2.4(a). Increasing groups reduces the spread of layers which decreases inter-layer and intra-layer communication, and providing higher throughput. Beyond a certain point though, it is difficult to create groups with uniformly equal work, resulting in resource under-utilization and lower throughput. Throughput is therefore optimal for a group count that varies between 2 to 8. This is the case only for data parallelism as it is computation-bound, while the communication-bound model parallelism keeps improving by using more groups.

  For design points where the number of groups is equal to the number of chips (16C-16G, 32C-32G, etc.), no inter-chip partitioning is required due to which data and model parallelism behave similarly, yielding similar throughput.

- **Impact of batch size:** The results in Figure 2.4(a) align with those of prior works. At the larger batch size shown in Figure 2.4(b), we see significant deviation. A larger batch size increases the sizes of activation/error tensors. Data parallelism incurs communication of activations/errors only at group boundaries, i.e., inter-group communication. For large batches, inter-group communication becomes a significant bottleneck, adversely impacting performance. Hence, for large batches, data parallelism achieves the best performance with a group size of 1, i.e., a PipeDream pipeline is ineffective and the execution reverts to a fully temporal pipeline. Figure 2.5 shows how the performance varies with batch size for a fully temporal pipeline (1 group) and pipeline-parallel training (16 groups). We observe that a PipeDream spatial pipeline is not effective beyond a batch size of 512.

- **Impact on energy efficiency:** Energy to train ResNet-50 with different configurations at a batch size of 8192 is shown in Figure 2.6. Model parallelism consumes significantly more energy than data parallelism due to the higher intra-layer and inter-layer communication. Increasing groups benefits model parallelism highly as it reduces inter-worker communication energy. The benefits of increasing groups is trivial for data parallelism as gradient communication accounts to a small fraction of

overall energy, and reducing that doesn't translate to significant energy savings. We observe a similar pattern for smaller batch sizes as well.

Observation-1: *For convolutional networks, pipeline-parallel training performs best at small batch sizes and usually prefers small-moderate number of groups. Fully temporal pipelines (number of groups = 1) performs best when the batch size is large.*

2.3.2.2 Fully Connected Networks

Next, we analyze how batching, grouping, and partitioning affects training throughput of the other major class of DNNs - fully connected networks. Figure 2.7 shows training throughput for Bert-base.

- **Impact of Batch Size:** Unlike the analysis for ResNet-50, we observe that behavior for Bert-base is very different for batch sizes of 128 and 8192. As discussed earlier, and as observed by prior works, model parallelism requires communication of activations across workers whereas data parallelism has to aggregate gradients across workers. We observe that at batch size of 128, shown in Figure 2.7(a), model parallelism performs better than data parallelism at all design points. At a batch size of 128, as Bert-base has more weights than activations, model parallelism performs less communication and therefore achieves better throughput. Figure 2.8 quantifies model and activation sizes of Bert-base at different batch sizes. Data parallelism incurs only gradient communication, whose size is independent of batch size, and model parallelism can incur both intra-layer and inter-layer communication, which is directly proportional to batch size. For batch sizes beyond 1024, activations/errors become dominant, returning the advantage back to data parallelism (as seen in Figure 2.7(b). We see a similar trend for energy as well in Figure 2.9. At small batch sizes, the execution is bottlenecked by memory accesses. This motivates moving to larger batch sizes until the execution is compute-bound. We see an order of magnitude improvement in throughput in Figure 2.7 as we move from a batch size of 128 to 8192. Thus, for high throughput, Bert-base must be executed with a large batch size which makes its trend lines similar to that for ResNet-50 (as we'll further discuss).

  Figure 2.10 shows the throughput for data and model parallel training of Bert-base

at different batch sizes. Model parallelism performs better at small/moderate batch sizes and data parallelism performs better at large batch sizes, eventually plateauing at high compute utilization. The cross-over point occurs for Bert-base at a batch size of 512. This crossover point varies for different DNNs, dictated by the batch size and the DNN's neuron-to-weight ratio.

- **Impact of Grouping:** Both data and model parallelism are limited by data movement at smaller batch sizes – data parallelism due to gradient aggregation and model parallelism due to inter-layer communication. Therefore, increasing the number of groups reduces inter-worker communication for both data and model parallelism as layers get mapped to fewer workers, thus increasing throughput and reducing energy.

  For larger batches, data parallelism training becomes compute-bound. More groups is helpful at reducing communication at first, but then with a large number of groups, load imbalance is more noticeable. As shown in Figure 2.7(b), optimal throughput is observed when the group count is 4 to 16, a trend similar to that for ResNet-50.

Observation-2: *For fully connected networks, a large batch size is required to alleviate the memory and communication bottleneck; at large batch size, the trend lines are similar to that for convolutional networks, with a medium group count achieving optimal throughput. If the algorithm or memory capacity demands a small batch size, model parallelism may end up out-performing data parallelism depending on the DNN's neuron-to-weight ratio.*

2.4 Cafine

In our exploration, PipeDream's pipeline-parallel approach with 4-16 layer groups is often the best. This section performs a deeper dive on this configuration, further expanding the design space, to uncover additional energy savings.

2.4.1 Chip-Level Execution Strategies

At the cluster-level, we retain PipeDream's spatial pipeline, splitting the DNN into layer groups and assigning a set of nodes to each layer group. Within a layer group, PipeDream and our analysis so far has assumed a temporal pipeline.

When the chip has a tiled architecture like SIMBA or SCALEDEEP (instead of a monolithic design as in the Google TPU), there is an opportunity to execute layers in a spatial pipeline within a chip to potentially reduce data movement. To comprehensively evaluate the design space, we consider multiple implementations for such a spatial pipeline.

A typical spatial pipeline example is shown in Figure 2.2(e). We consider a naive spatial pipeline (NSP) implementation where a layer has to finish before its outputs can be consumed by the next layer. While simple to implement, it can lead to more memory accesses if a layer's output activations/errors exceed the on-chip buffer sizes. We also consider a fine-grained spatial pipeline (FG-Pipe), where the next layer is initiated as soon as the previous layer has generated a large enough set of activations/errors [7, 111]. This increases reuse within buffers during the forward and backward passes, but still requires forward-pass activations to be written to memory so they can be reused during the backward-pass. FG-Pipe reduces the energy consumption by reducing memory and interconnect overheads. It doesn't provide any performance benefits, and suffers from load imbalance like NSP.

2.4.2 Cafine: Criticality-Aware Fine-Grained Pipeline

We now evaluate different chip-level approaches for the layers within a PipeDream group. The subsequent figures assume 64 chips, 8 groups, and batch size of 512; they contrast PipeDream (temporal pipeline within each chip) and FG-Pipe. Figure 2.11 shows the latency for each group with the temporal and spatial pipelines for ResNet-50. The FG-Pipe latency for each group is always higher than the latency in the baseline PipeDream's chip level temporal pipeline. Note that the overall throughput of PipeDream's cluster-level spatial pipeline is ultimately determined by the latency of the slowest group, in this case, group 6. The FG-Pipe design is 5% slower than PipeDream in Group-6, thus yielding 5% lower throughput.

Meanwhile, Figure 2.12 shows the energy consumed by each group with these two pipelines and we see the opposite effect - FG-Pipe consumes less energy than PipeDream for each group because it promotes higher reuse in on-chip buffers.

This basic performance-energy trade-off can be exploited by PipeDream's cluster-level spatial pipeline. We propose Cafine, which treats critical and non-critical groups differ-

ently. The critical group (group 6 in this case) is executed with a temporal pipeline so that the pipeline's throughput is the same as the baseline PipeDream. The non-critical groups are executed with FG-Pipe - while each group experiences a higher latency, that latency is lower than the latency of critical group 6. But each of these non-critical groups dissipates lower energy than the baseline PipeDream. The red circles drawn in Figures 2.11 and 2.12 show the design points chosen by Cafine for each group, yielding the best of both approaches.

2.4.3 Results

Next, we evaluate Cafine on a variety of vision and NLP models and compare it against different baselines: a Fully Temporal Pipeline (FTP) resembling AccPar, PipeDream, Naive Spatial Pipeline (NSP), and Fine-Grained Spatial Pipeline (FG-Pipe). For every DNN, Cafine and the baselines are evaluated at their optimal partitioning, mapping, and grouping strategies. The system considered is a medium-scale cluster with 64 chips. Throughout this section, CNNs are evaluated at a batch size of 512, beyond which the optimal pipelining strategy is FTP and not pipeline-parallelism, as discussed in Section 2.3.2.1. NLP models on the other hand are evaluated at a batch size of 8192. As mentioned in Section 2.3.2.2, pipeline-parallel training is better than FTP at all batch sizes for FC models, and large batch training provides better throughput.

2.4.3.1 Convolutional Networks

Figure 2.13 plots both throughput and energy for the various design points for ResNet-50; note that the axes don't start at zero to more clearly show the trade-offs in these metrics. PipeDream has the highest performance while a fully temporal pipeline has the least energy. FG-Pipe is 5-6% lower than PipeDream in terms of energy and throughput. Cafine bridges that gap, matching the throughput of PipeDream and consuming 4% less energy than PipeDream. The fully temporal pipeline has 2% lower throughput and 12% lower energy than Cafine. The energy differences between FG-Pipe, PipeDream, and Cafine are primarily because of buffer overflow/reuse and memory/interconnect accesses as discussed earlier. The fully temporal pipeline incurs just gradient communication, whereas Cafine requires inter-group communication of activations/errors. As the volume of gradients is considerably less compared to activations/errors in ResNet-50, FTP con-

sumes the lowest energy. But as gradient aggregation in FTP cannot be overlapped, it has poor throughput compared to Cafine.

2.4.3.2 Fully Connected Networks

Next, in Figure 2.14, we consider throughput and energy for Bert-base, where we adopt data parallelism. Unlike ResNet, where the fully temporal pipeline has the least energy, here it has the highest energy. This is because of the high data movement costs for gradients (which is significantly larger in Bert-base than ResNet-50) as each layer is spread across the entire cluster. Thus, interconnects account for 76% of total energy for FTP. This also results in slowdown for FTP. Meanwhile, both Cafine and FG-Pipe consume 11% less energy than PipeDream, while matching its throughput. The performance-energy trade-off is less stark in Bert-base because of its structured nature, which leads to better load balance among layers within a group.

2.4.3.3 Discussion for All Workloads

Figure 2.15 shows the throughput and energy, relative to PipeDream, for each workload and execution strategy. For FC models, Cafine has zero performance loss and consumes an average of 8% less energy compared to PipeDream. For CNNs, Cafine consumes 4% less energy on average than PipeDream with no loss in performance. The fully temporal pipeline consumes 6.5% less energy than Cafine for CNNs, but it is also 12.5% slower. On average over different CNN and FC workloads, Cafine achieves the best performance – on-par with PipeDream – while consuming the least energy (6% less energy than PipeDream on average). Compared to NSP and FG-Pipe, Cafine consumes 7% and 1% less energy while being 12% faster.

So far, we have seen different strategies being optimal for different workloads under different scenarios. No single execution strategy is optimal for all cases. In Table 2.2 we list the best execution strategy for each DNN class (CNN or FC) at different batch sizes, depending on the priority metric (performance, energy, or EDP). We envision a future tool that can integrate our analytical performance/energy model, our design space exploration, the DNN characteristics, and the hardware configuration to estimate the optimal execution strategy. The analysis in this chapter is a first step towards such a tool.

2.4.3.4 Discussion on Memory Requirements

We close our analysis with a look at another important metric in DNN training - the off-chip memory requirement. While Cafine and pipeline-parallel training provide higher performance than fully temporal pipelines, they incur a memory overhead. PipeDream requires as many weight copies as there are groups to ensure high accuracy. Figure 2.16 shows the memory requirement for activations and weights, and the performance of Cafine, relative to a fully temporal pipeline. While Cafine has a similar memory requirement as PipeDream, it has a steep memory requirement cost compared to FTP, somewhat proportional to its speedup (and energy efficiency).

2.5 Related Work

Many works explore optimal dataflow, tiling, and partitioning for DNN training at both intra-chip and inter-chip levels.

Intra-Chip Techniques. TimeLoop [97] and Maestro [70] provide a systematic approach towards designing accelerators and dataflows as loop nest representations. They entail an optimizer-guided cost model to evaluate the energy and performance of different design points. Interstellar [137] extends Halide [103], a domain-specific language and compiler, to represent different architectures and dataflows as Halide schedules. They aim to find the optimal loop ordering, blocking, and parallelization for a workload. Instead of performing an exhaustive search, GAMMA [60] uses a genetic algorithm to find efficient mapping strategies. ConficiuX [59] uses a hybrid reinforcement learning and genetic algorithm to find optimal resource allocations per layer. Much of the above work is focused on DNN inference and limited to a single chip.

Inter-Device Techniques. OWT [67] proposes data parallelism for convolutional and model parallelism for fully-connected layers. HyPar [117] uses an analytical approach to decide the optimal layer partition. AccPar [118] extends HyPar with an exhaustive search of layer-wise tensor partitions. AccPar also applies to heterogeneous systems, and DNNs with multi-path topologies. OptCNN [53] extends the partition space to include height/width of feature-maps and uses dynamic programming. FlexFlow [54] improves upon OptCNN by considering all types of networks and partitions, and by using a Markov Chain Monte Carlo algorithm.

Table 2.1. Chip specifications.

Component	Spec	Energy (pJ/bit)	Area(mm^2)
Input buff	16KB	0.028	0.12
Weight buff	64KB	0.048	2.88
Psum buff	4KB	0.026	0.19
Vector MAC	8-MACs	4.4 pJ	0.33
Global buff	512KB	0.12	2.4
Inter-tile	82GB/s	0.47	-
Inter-chip	64GB/s	2.87	-
HBM	128GB/s	4	-

Table 2.2. Optimal execution strategies for CNN and FC workloads based on batch size and priority metric – performance, energy efficiency, EDP. DP/MP are data/model parallelism.

Workload	CNN		FC	
Batch size	Small/Med	Large	Small/Med	Large
Perf	Cafine+DP	FTP+DP	Cafine+MP	Cafine+DP
Energy	FTP+DP	FTP+DP	Cafine+MP	Cafine+DP
EDP	Cafine+DP	FTP+DP	Cafine+MP	Cafine+DP

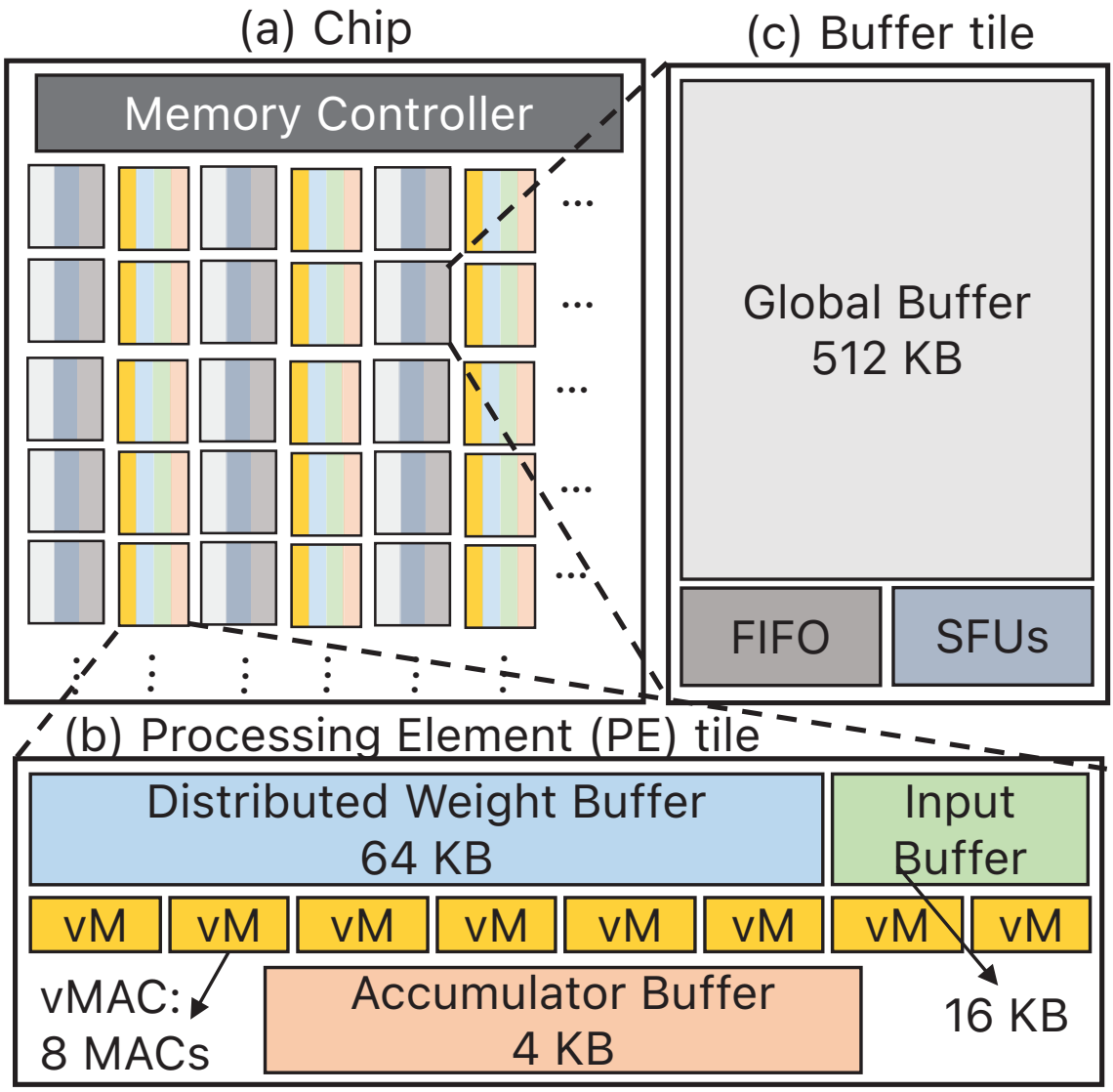

Figure 2.1. Chip (a), PE tile (b), and buffer tile (c) architecture.

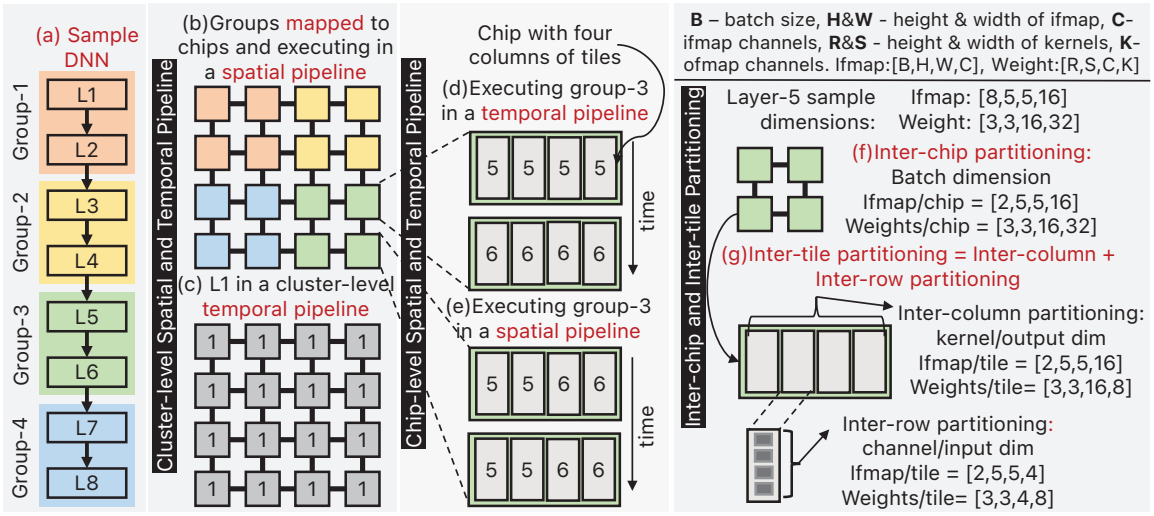

Figure 2.2. Illustration of different components of execution strategy for an example DNN, cluster, chip, and layer.

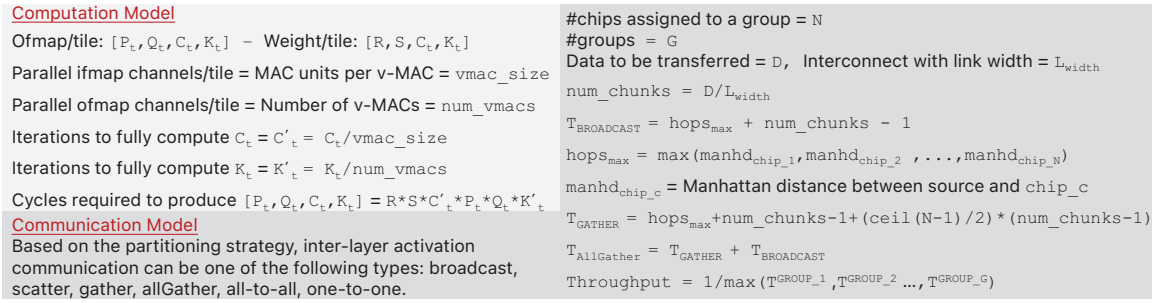

Figure 2.3. Brief overview of the performance model used in the analytical simulator.

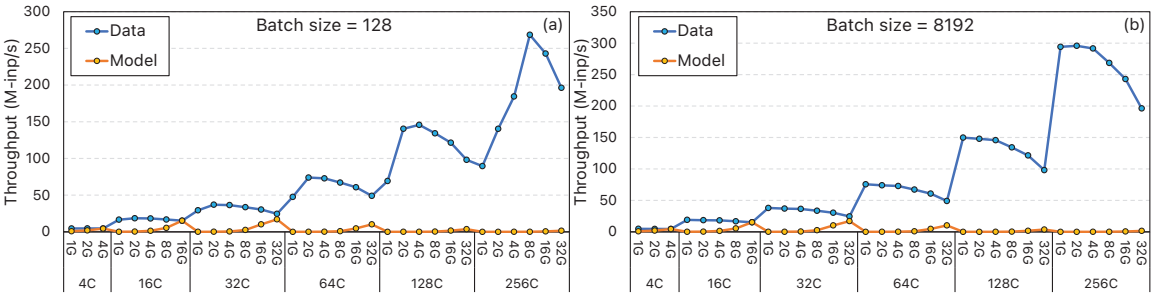

Figure 2.4. Exploring the design space of ResNet-50 training by varying number of groups, number of chips, batch size, and partitioning strategy. xC refers to a design with x chips and xG refers to a config where ResNet-50 is partitioned into x groups.

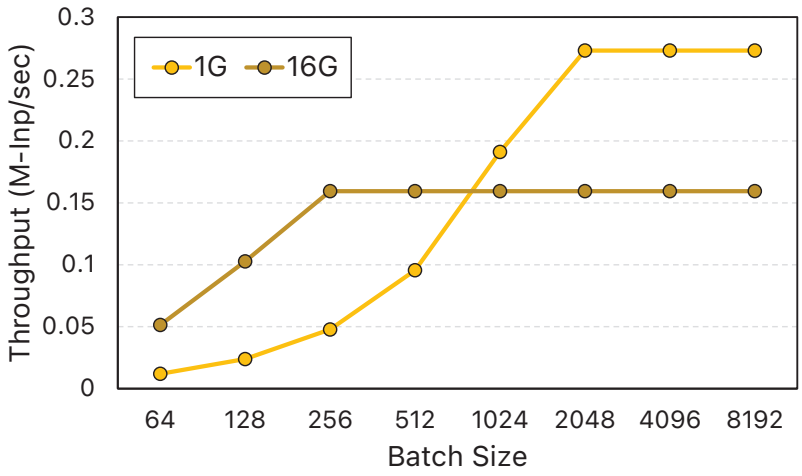

Figure 2.5. Throughput of data parallel training of ResNet-50.

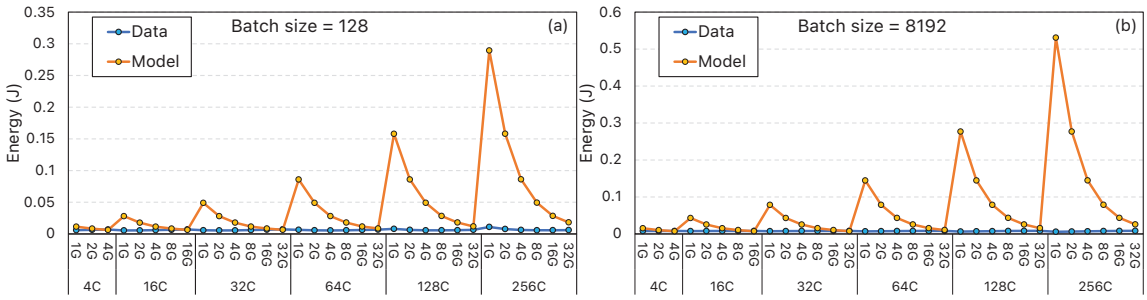

Figure 2.6. Energy efficiency of ResNet-50 for different configurations.

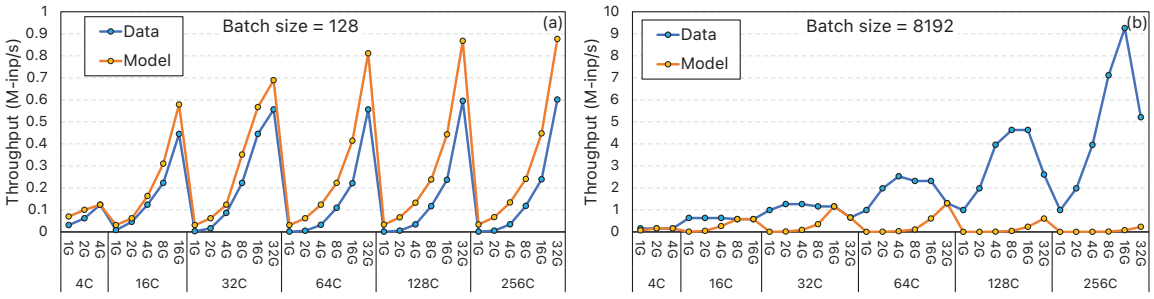

Figure 2.7. Exploring the design space of Bert-base training by varying number of groups, number of chips, batch size, and partitioning strategy. xC refers to a design with x chips and xG refers to a config where Bert-base is partitioned into x groups.

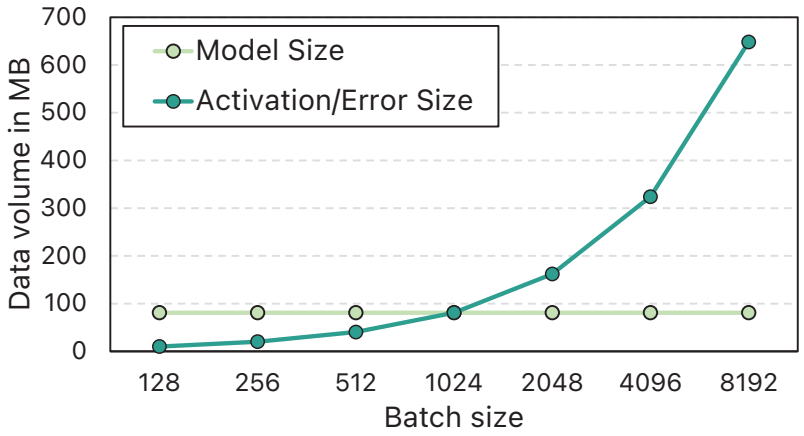

Figure 2.8. Size of activation/error and weight tensors of Bert-base at different batch sizes.

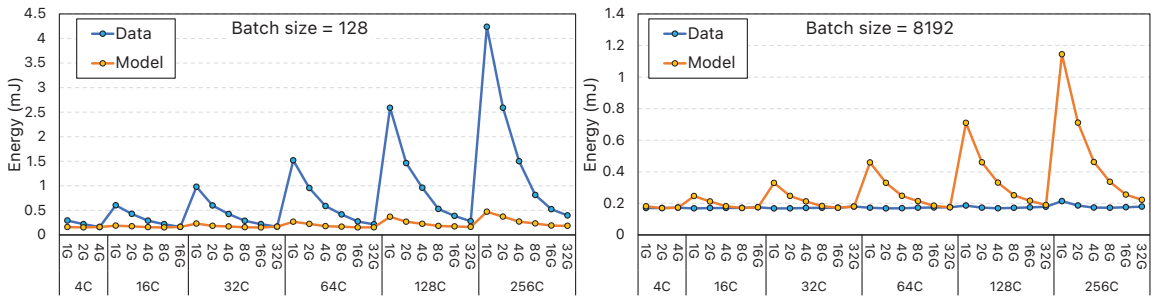

Figure 2.9. Efficiency of training Bert-base for different configurations.

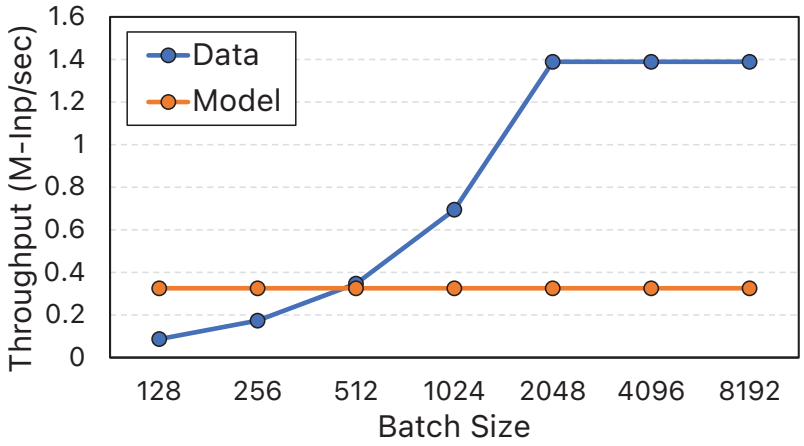

Figure 2.10. Bert-base throughput for data/model parallelism.

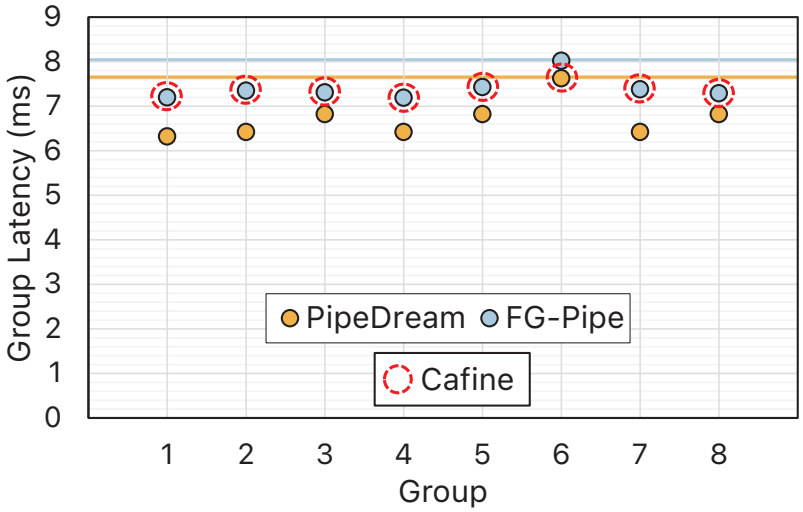

Figure 2.11. Execution latency of all groups in ResNet-50 for PipeDream and FG-Pipe. Red circle denotes the execution strategy chosen for the specific group by Cafine.

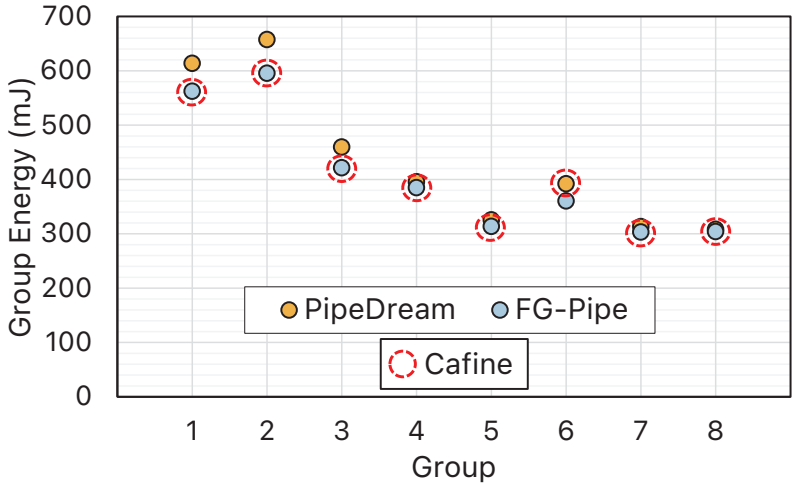

Figure 2.12. Group-wise energy consumption of ResNet-50 for PipeDream and FG-Pipe. Red circle denotes the execution strategy chosen for the specific group by Cafine.

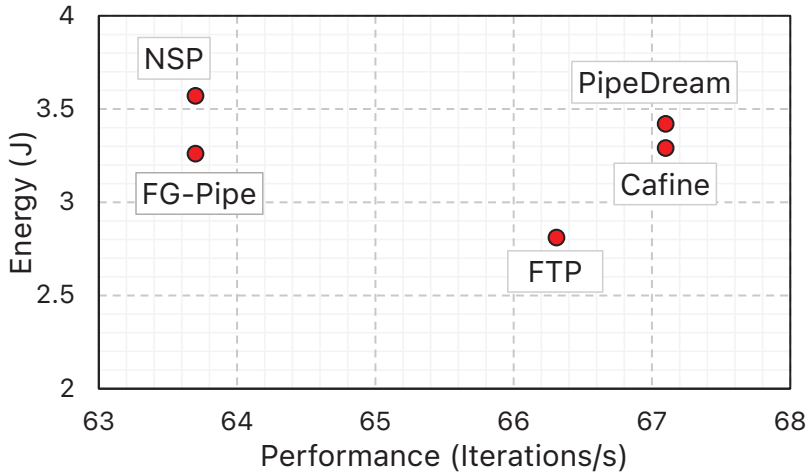

Figure 2.13. Chip level execution strategies for ResNet-50.

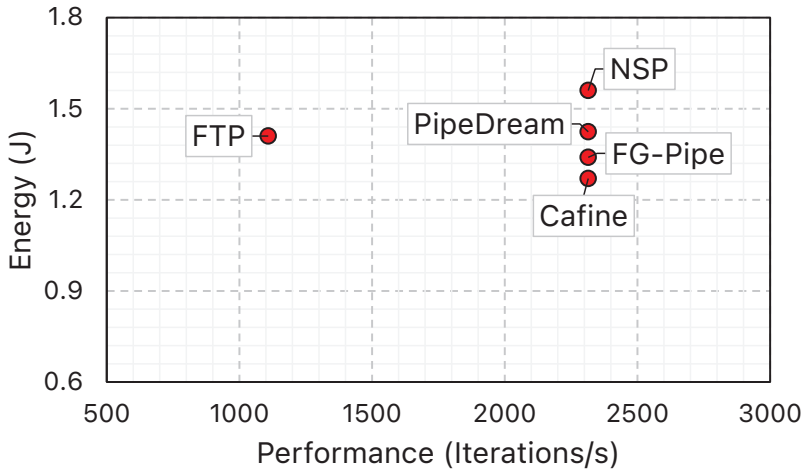

Figure 2.14. Chip level execution strategies for Bert-base.

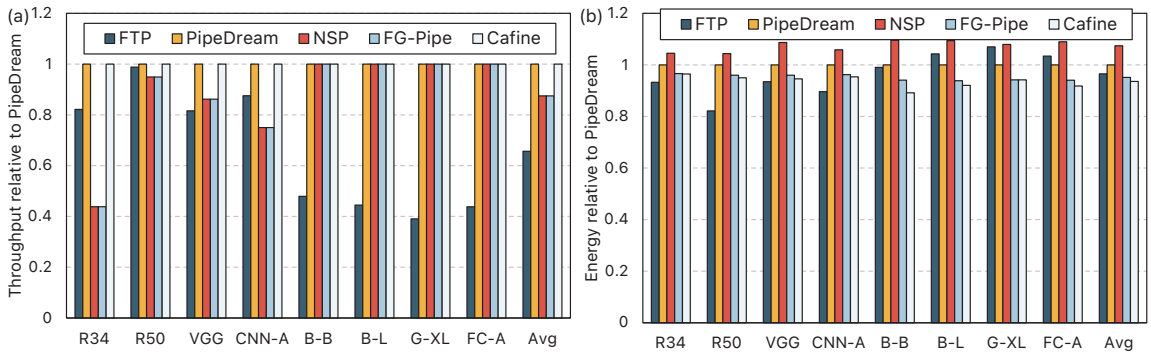

Figure 2.15. Comparing with PipeDream. (a) Performance and (b) Energy of different execution strategies relative to PipeDream. **CNN-A**: Average across CNN workloads (**ResNet-34**, **ResNet-50**, **VGG-D**), **FC-A**: average across FC workloads (**Bert-base**, **Bert-large**, **GPT2-XL**), **Avg**: average across all the six workloads.

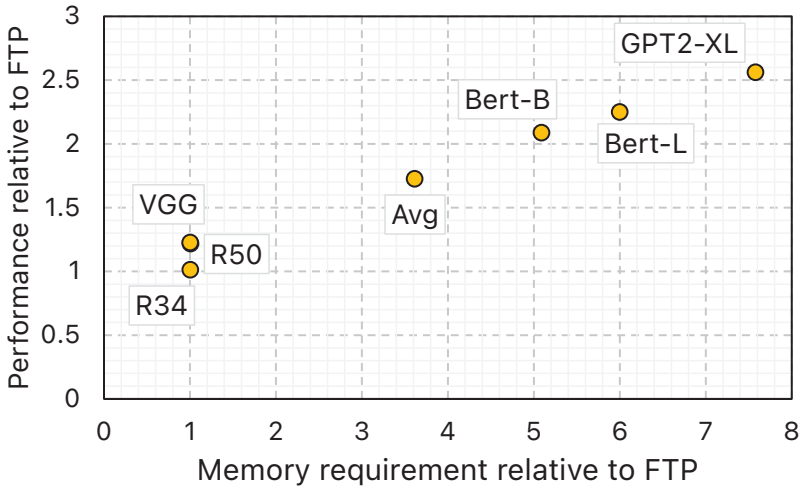

Figure 2.16. Performance and memory requirements of Cafine compared to a fully temporal pipeline.

CHAPTER 3

INXS: BRIDGING THE THROUGHPUT AND ENERGY GAP FOR SPIKING NEURAL NETWORKS

3.1 Introduction

The stagnation of Moore's Law scaling has shifted industry and academia's focus away from general-purpose systems and towards specialized systems. Neuromorphic architectures are an important class of specialized systems because:

1. They are efficient at a variety of machine learning tasks that are growing in prominence – image analysis in self driving cars, information discovery from massive datasets, etc.

2. They target the grand challenge of emulating brain mechanics in hopes of matching the brain's cognitive power and energy efficiency.

These two separate needs – machine learning efficiency and brain emulation – have also led to a bifurcation in neuromorphic architectures.

A number of architectures, DaDianNao [16], ISAAC [111], EIE [112], Cnvlutin [5], and Eyeriss [15] to name a few, are based on the artificial neurons (perceptrons) that have formed the basis for decades of research in machine learning. We refer to these architectures as artificial neural network accelerators, or *ANN accelerators*.

The second class of architectures, TrueNorth [4], SpiNNaker [62], and Neurogrid [8] to name a few, are based on biologically plausible models of spiking neurons. We refer to these architectures as spiking neural network accelerators, or *SNN accelerators*.

The goal of this chapter is not to compare ANN vs. SNN accelerators. To date, only one study, by Du et al. [28], has performed a head-to-head comparison of ANN and SNN accelerators. While that study is an excellent start to an important debate, it draws lim-

ited conclusions for small-scale chips executing small-scale networks. For example, they conclude that SNNs achieve significantly lower accuracy than ANNs on MNIST, but concurrent work [31, 32] developed better training algorithms for SNNs and achieved 99.42% accuracy on MNIST. Du et al. investigate chips with no more than 110 artificial neurons and 300 spiking neurons. We mention these examples to highlight that the comparison between ANNs and SNNs is far from being resolved, and will likely play out over the coming decade.

In the meantime, advances are required for both ANNs and SNNs. At the moment, much of the architecture research has focused on ANNs. As a result, ANNs are ahead of SNNs on a variety of metrics (see Table 3.1). This chapter attempts to bridge that gap by designing better SNN architectures that can keep up with the high throughput and energy efficiency being achieved on state-of-the-art ANNs.

The most high-profile and most efficient SNN architecture to date is IBM's TrueNorth. It is a 5.4 billion transistor chip that can model 1 million neurons and 256 million synapses while consuming less than 100 mW. TrueNorth achieves high tile-level parallelism, and makes a number of design choices that impose constraints on the application, while reducing power and storage requirements. However, we see in Table 3.1 that TrueNorth lags behind state-of-the-art ANN accelerators on all metrics, notably throughput and energy. Therefore, drawing inspiration from recent ANN architectures, we undertake an overhaul of the TrueNorth design.

We describe an SNN accelerator that leverages memristor crossbars to aggregate the effects of input spikes in the analog domain. By effectively using in-situ computing, memristor crossbars have been shown to achieve high parallelism and storage density in the ISAAC [111] and PRIME [18] ANN accelerators. We describe the many changes required to adapt a crossbar-based architecture for an SNN. In particular, the management of neuron potentials represents the biggest challenge, and the sparse spike rate represents the biggest opportunity. The former requires additional storage overheads, and the latter enables low overheads for analog-to-digital conversion (ADC). We carry out a design space exploration to identify the best provisioning of resources for this mixed-signal architecture.

3.2 Background

Artificial neurons were developed more than 70 years ago [83]. Artificial neurons receive synchronous real-valued inputs, perform a dot-product of these inputs with weights, apply an activation function (often ReLU), and pass real-valued outputs to the next layer of artificial neurons. In addition to many decades of progress, the past decade has seen significant advances with artificial neurons, primarily because of our ability to train deep networks with a combination of new techniques.

3.2.1 Spiking Neurons

While scientists have delved into the mechanics of the biological neuron for decades [46], it has only recently received attention from the architecture community. A number of high-profile projects [4, 8, 62] have attempted to implement biologically plausible neuron models in hardware. Many of these hardware projects implement neuron models that are highly simplified, but that can emulate many biologically observed neuron behaviors, e.g., the Izhikevich neurons [94].

The most popular of these simple neuron models is the Linear Leaky Integrate and Fire model (LLIF), shown in Figure 3.1. An LLIF neuron is stateful – in addition to synaptic weights, it retains the value of its (membrane) potential. This potential reflects inputs that have been received in the recent past. Inputs are received in the form of binary spikes. When a spike is received on an input, the synaptic weight for that input is added to the potential (see Figure 3.1). In every cycle, a leak is also subtracted from the potential. When the neuron's potential eventually reaches a specified threshold, the neuron produces an output spike of its own. After the spike, the neuron potential is reset.

Spiking neurons have the potential to be hardware-efficient because inputs and outputs are binary spikes, i.e., a communication link between neurons requires a single bit. Further, the spiking neuron model does not require a multiplier – because the input is binary, the synaptic weight is simply added to the potential. Spikes can therefore lead to efficient communication and computation.

Because a neuron is designed to respond after observing spikes over time, the input is provided over an *input interval*, say 500 cycles. Figure 3.2 shows how each pixel of an input image is converted into a spike train that extends across an input interval. These

spike trains are fed as inputs to the first layer of neurons. Prior work has primarily used *rate codes* that convert an input pixel value into a certain number of spikes. For example, a red pixel value may be converted into 50 evenly spaced spikes in 500 cycles, while a blue pixel value may be converted into 125 evenly spaced spikes in the input interval. The same encoding is typically used throughout the network, i.e., information is carried in terms of spike intensity. The code also includes an element of stochasticity, e.g., a rate code typically uses a Poisson distribution to inject spikes [3].

Spiking neurons are typically trained with a biologically plausible process called STDP (Spike Timing Dependent Plasticity [34]). This is an unsupervised training method where each neuron adjusts its weights based on a local process. Recent studies have been unable to achieve high accuracies with STDP-based training [28, 110]. Therefore, more recent works have resorted to supervised backpropagation-based training for spiking networks [31, 32].

3.2.2 SNN Accelerators

IBM's TrueNorth processor [85] is the most prominent example of a digital architecture for large SNNs. We will use TrueNorth as the SNN baseline in this work because it achieves best-in-class throughput and energy efficiency.

TrueNorth is composed of many tiles, where each tile implements 256 neurons, each with 256 inputs. The tiles communicate through an on-chip and potentially an off-chip network. The tiles use a mix of asynchronous and synchronous circuits to boost energy efficiency. In every 1ms *"tick"*, a tile processes all received input spikes; any resulting output spikes are sent through the network to neurons in the next layer so they can be processed in a subsequent tick. TrueNorth implements an LLIF neuron model with a number of configurable parameters, including some that allow stochastic behavior. Within a tick, the tile sequentially walks through every neuron in that tile and every input spike to perform several updates to each neuron potential. For each neuron, it reads a 410-bit SRAM row that contains all parameters for that neuron, including a 256-bit vector indicating which tile inputs connect to that neuron. This bit vector is reconciled with the list of input spikes in that tick to identify spiking connections for that neuron. The synaptic weight for each of these connections is then sent to a synchronous neuron unit

that performs the necessary arithmetic operations. This unit adds the synaptic weights to the neuron's potential. Finally, the leak is subtracted and the potential is compared against the threshold. In case of an output spike, the neuron potential is reset. The final neuron potential is then written back to the SRAM bank. The 12.8 KB SRAM bank occupies nearly half the tile area and one-third the tile power. A tile processes a single synapse at a time. The tick is long enough (1ms) to process all possible input spikes and neurons sequentially.

To further reduce storage requirements and energy, TrueNorth imposes several constraints on the neural network. It only uses 4 quantized 9-bit weights per neuron. It also forces an input spike to share the same weight type with all neurons in that tile. A neuron's output can only be seen by the 256 neurons in one tile. A neuron can only receive at most 256 inputs.

SpiNNaker [62] is another prominent SNN architecture that uses many low-power general-purpose ARM cores to perform several parallel neuron updates. It is well known that custom ASICs will out-perform general-purpose cores by at least two orders of magnitude [42], so we will not explore SpiNNaker-style architectures in this chapter.

A few projects have attempted to implement neurons and synapses with analog devices, typically using capacitors or memristors to emulate neuronal behavior [4, 77, 78, 124]. These projects have focused more on device innovations to reproduce neuron behavior, and have not focused on architectural innovations to boost throughput. For example, Liu et al. [78] implement a single 32×64 memristive crossbar to execute feedforward and Hopfield networks. The crossbar performs the synaptic operations, and an analog integrate-and-fire circuit models the neuron. But, maintaining the neuron potential in an analog circuit can incur a very high area overhead, especially in large-scale convolutional networks where the number of neurons far exceeds the number of (shared) synapses. However, we do believe that analog circuits have a lot to offer [55] and we will use the analog domain in a limited manner to accelerate the neuron update.

3.2.3 ANN Accelerators

Our proposed architecture is inspired by the best practices in state-of-the-art ANN accelerators. We first discuss the analog approach, followed by the digital approach.

Two architectures introduced in the past year, ISAAC [111] and PRIME [18], have lever-

aged memristor crossbars to perform dot product operations in the analog domain and accelerate deep convolutional neural networks. We will focus on ISAAC here because it out-performs PRIME in terms of throughput, accuracy, and ability to handle signed values. We note that a few other works have also analyzed the circuits required in crossbar-based accelerators [22, 122, 134].

A memristor crossbar uses Kirchoff's Law to produce a sum of products, as shown in Figure 3.3. Inputs are provided as a vector of voltages; the memristor conductances in the crossbar represent synaptic weights of neurons; the emerging bitline currents are neuron outputs (before the activation function) because they represent the dot products of input voltages and synaptic weights. This is an example of in-situ computing because the crossbars are not only used to store weights, but also perform computations on them. ISAAC uses a number of crossbars in a tiled architecture to process all layers of a deep network in parallel. It distributes computations across time and space to manage the high costs of analog-to-digital conversion (ADC). Even with such techniques, the ADCs account for a large fraction of chip power and area. To support sufficient precision, ISAAC employs 8-bit ADCs to capture the largest possible dot-product emerging from a crossbar bitline. The dot products, after analog to digital conversion, are aggregated with digital ALUs. eDRAM banks are used to store neuron outputs until they are consumed by the next layer. By setting up a pipeline from layer to layer, a relatively small set of outputs has to be buffered, which can be accommodated in a 64 KB eDRAM unit per tile. Since a dense crossbar is used to store the weights and perform computation, ISAAC is able to dramatically reduce data movement, and increase computation/storage density.

We next describe recent digital ANN architectures. The DianNao [13] and DaDianNao [16] accelerators were among the first to target deep convolutional networks. DianNao designs the digital circuits for a basic NFU (Neural Functional Unit) that can process 16 inputs to 16 neurons in parallel. DaDianNao is a tiled architecture where each tile has an NFU and eDRAM banks that feed synaptic weights to that NFU. DaDianNao uses many tiles on many chips to parallelize the processing of a single network layer. Once that layer is processed, all the tiles then move on to processing the next layer in parallel. Thus, the keys to DaDianNao's efficiency are: (i) localized data movement (from local eDRAM bank to nearby NFU), and (ii) time-multiplexed execution of several neurons

and several network layers on a small set of SIMD execution units (the NFUs). Other recent works have proposed innovations to digital ANN accelerators that primarily exploit sparsity [5, 106, 112]. Since digital ANN accelerators are nearly an order of magnitude slower than ISAAC [111], we will not consider them further in this chapter.

3.3 The INXS Architecture

3.3.1 Overview

[120]As described in the previous section, the best SNN accelerator to date, TrueNorth, suffers from a few weaknesses:

1. There is no intra-tile parallelism while performing neuron updates.
2. Each input spike to a neuron is handled sequentially.
3. To reduce the storage and energy overheads, significant approximations have to be made for the synaptic weight values.
4. A neuron can only have at most 256 inputs and its output can be seen by at most 256 other neurons connected to one axon.

We design a mixed-signal architecture, INXS[1], that addresses all of the above problems, and can efficiently handle state-of-the-art deep networks. A large number of memristor crossbars are used to process the many incoming spikes in a tick, and compute the resulting potential increments in parallel. The potential increments are immediately converted to digital signals. The neuron potentials are retrieved from SRAM buffers with wide reads, added to the increments, thresholded, and written back to SRAM. The resulting spikes are routed to the next layers so they can be processed in the next tick. Many crossbars work in unison on different layers of the neural network to set up an efficient pipeline.

The key contributions of this design are:

1. It offers very high pipelined parallelism with many crossbars, not only working on many SNN layers in parallel (as in TrueNorth), but also working on many neuron update values and many input spikes in parallel (unlike TrueNorth). In most cases, the

[1]INXS, pronounced “in excess” is short for IN-situ Xbar Spiking

pipeline operates as an odd-even pipeline, working on analog crossbar operations in odd ticks, and digital neuron updates in even ticks.

2. While some prior works [78] have implemented a crossbar in tandem with analog neurons, we observe here that in a convolutional network, a set of shared weights are used to compute several neurons. The use of analog neurons would require a single crossbar bitline to be multiplexed across many analog circuits, resulting in significant overheads. Therefore, we immediately convert the analog crossbar output into a digital signal and perform the even phase in the digital domain.

3. We lay out several design details and carefully consider the overheads of each module. We follow with a design space exploration to identify how best to provision the resources per tile.

4. The resulting architecture differs from the state-of-the-art ANN accelerator, ISAAC, in the following ways: (i) ISAAC requires a 22-stage pipeline while INXS only requires a 2-stage pipeline to process a single neuron in one layer, (ii) INXS uses a low-resolution ADC because of observed sparsity, thus achieving higher throughput per area, and (iii) it allocates more area for central storage and neuron update.

5. The resulting architecture differs from the state-of-the-art SNN accelerator, TrueNorth, in the following ways: (i) INXS does not constrain neuron input/outputs and weights in any way, (ii) it offers orders of magnitude higher parallelism and throughput, and (iii) it achieves lower energy per operation by boosting throughput and lowering the contribution of leakage.

3.3.2 Implementation Details

3.3.2.1 Overall Chip Organization

INXS is designed to be modular and hierarchical. Figure 3.4 shows that a chip is composed of several tiles connected with a mesh network. The many layers of an SNN are scattered across these tiles. Figure 3.4 also zooms into one of these tiles. A tile has *central SRAM buffers*, *Neuron Units*, *Synaptic Units*, and a router. The Synaptic Units, Neuron Units, and the router in a tile are connected by a unidirectional ring network. The ring network has North, South, East, West, and Hub stations – the Hub is used to switch to

the inter-tile mesh network. Figure 3.5 shows the details of a Synaptic and Neuron Unit in a tile. Each Synaptic Unit is composed of multiple memristor crossbars and ADCs. The Neuron Unit has the adders and thresholding logic to implement the neuron model. Next, we'll walk through the operations required to execute a single convolutional layer.

3.3.2.2 The Odd Phase

In every odd tick, all the crossbars on the chip receive inputs from their input buffers. A tick is assumed to be at least 100 ns [111] to allow sufficient time to perform a crossbar read and capture all the bitline outputs in sample and hold circuits. This phase exploits very high parallelism in the analog domain to estimate the effect of every incoming spike on the potential of several neurons.

We'll assume that a crossbar has R rows and C columns of w-bit cells. We'll assume that weights and neuron potentials are stored with p-bit fixed-point precision. Depending on the values of p and w, a single synaptic weight may be spread across multiple cells in a row. If a neuron has more than R inputs, its calculation will be spread across multiple crossbars, and potentially multiple Synaptic units.

3.3.2.3 The Even Phase

The even phase is itself composed of several small *"cycles"*. In the first cycle, the ADC processes the first bitline output. The results of multiple bitlines (after ADC) have to be aggregated with shift-and-add circuits in the Synaptic Unit because they represent contributions from different bits of the synaptic weights. If a neuron has more than R inputs and is spread across multiple crossbars, those partial results have to be aggregated as well. Once the partial result within a Synaptic Unit has been aggregated, this potential increment is placed on the output bus. The potential increment is routed to that neuron's home, possibly navigating 0 or more hops on the ring network and 0 or more hops on the mesh network. Once the partial sums are generated, the routing logic in each synaptic unit routes them to their respective neuron home through the ring bus if it resides in the same tile, or through the mesh network if the neuron home is in another tile. The partial sums are stored in the output buffer, which acts as input to the Neuron Unit.

Once the increment reaches the neuron home, it is added to the neuron's potential and leak in the Neuron Unit. To enable this addition, the neuron potential has to be read from

the central SRAM buffer in the previous cycle. Once the new neuron potential is calculated, it is thresholded, and the final neuron potential is written back to the SRAM buffer. The generated spike is then sent over the ring and mesh networks to a destination input buffer, where it will be accessed in the next Odd Phase.

All of these operations are performed deterministically and controlled by finite state machines in Synaptic and Neuron Units. The control signals for the finite state machines would be generated at compile time and loaded into the chip along with the weights for each network layer. We assume that the leak and threshold are the same for all neurons in a layer [9, 27]. We provision the network, SRAM buffer, and adders with sufficient bandwidth so they can handle the worst-case network layers in our evaluated workloads without introducing any structural hazards and contention. If such a chip had to evaluate an even larger network (say, more neurons or more inputs per neuron), it can do so, but would require multiple ticks to process the Even Phase.

Figure 3.6 shows the many pipeline stages that must be navigated for one neuron computation. At compile time, we would estimate the number of cycles required for one neuron increment, starting from the ADC, all the way until the resulting spike is placed in the next layer's input buffer. This number S will vary across layers and workloads depending on the required number of network hops. As each ADC sequentially walks through the C bitlines in its corresponding crossbar, new values begin navigating this S-stage pipeline. Therefore, the Even Phase will complete after $C + S$ cycles. The length of a tick is therefore variable across workloads, and is a function of the worst-case network layer in each workload.

Because consecutive network layers will typically map to adjacent tile quadrants or tiles, most communication in the network tends to be nearest-neighbor communication.

3.3.2.4 An Example Design Point

We carry out a design space exploration, where we vary a number of INXS parameters:

- Number of memristor xbars in a Synaptic Unit
- Number of Synaptic Units in a tile
- Number of Neuron Units in a tile

- Central buffer size

For each of these design parameters, we sweep through our example workloads and provision the bandwidths of each module so they can handle the worst-case layers. We then estimate the throughput, area, and power for each design point for our workloads. In Section 4.5, we show the design points that optimize throughput/area and throughput/power.

To make our architecture more concrete, we walk through the parameters selected for the optimal throughput/area design point, while assuming 16-bit fixed-point computations. This design uses 64 crossbars in a Synaptic Unit, and 8 Synaptic Units in each tile. The crossbar has 256 rows and 128 columns, and stores 2 bits per cell. The bus within a Synaptic Unit, the ring network, and the mesh network all have a width of 128 bits. Each of the 8 SRAM central buffers in a tile has a capacity of 128 KB, a row width of 128 bytes, and a read latency of 0.57 ns. The Neuron Unit contains 8 3-input adders and 8 comparators to perform the neuron activation.

3.3.2.5 Balancing the Pipeline

In one Odd/Even Phase, a convolutional kernel in a layer is applied to one set of inputs to produce one output neuron. This process has to be repeated over several Odd/Even Phases until the kernel has been applied to an entire set of input feature maps. Some layers have less work to do than others. Those layers can either idle in some cycles or we can replicate the weights and boost the throughputs of the work-intensive layers so every crossbar is busy in every cycle. Such replication leads to a balanced pipeline, similar to the one employed in ISAAC.

In our workload evaluations, we also observe that the spike rate is relatively sparse and that the maximum observed output from a bitline is significantly smaller than the worst-case output. While a 256x128 memristor xbar would need a 10-bit ADC to capture its worst-case output, we observe enough sparsity in our applications that an 8-bit ADC is actually sufficient. This significantly boosts the throughput/area and throughput/power metrics, while having zero impact on accuracy. We envision that a developer would have to run simulations to confirm that the ADC precision is rarely exceeded at run-time; if it is, such overflows can be avoided by mapping fewer inputs to every crossbar column.

3.3.2.6 Neuron Model

Note that we are implementing a simple LLIF neuron model. We are not modeling the many modes and stochastic features implemented by TrueNorth. The adder/thresholding unit can be augmented to handle these additional modes and we leave these as future work. It is worth noting that the adder/thresholding unit occupies 0.6% of tile area, so even if its size is increased by 10×, its overhead would be small. For this study, we assume that all pooling layers use average pooling, because average pooling is more amenable to crossbar acceleration than max pooling.

3.3.2.7 Routing Table

The outputs produced by bitlines of a crossbar are routed to the same neuron unit, and eventually to the same set of destination crossbars in the next layer. Each crossbar therefore has a single register that is used to route the result to its neuron home. The neuron home has a routing table that has one entry for each neuron. That entry keeps track of all the crossbars in the next layer that must receive the spike resulting from that neuron. We calculate the number of entries by analyzing the state-of-the-art neural networks like MSRA and VGG-NET. Based on our analysis, we fix the number of crossbars that each neuron output can connect to as 512 (128K neurons, 512× better than TrueNorth). We size these structures so they can handle the largest deep network to date; the resulting size of the routing table is 25.5 KB.

3.4 Methodology

We use the following metrics to evaluate the various design points:

- Computational Efficiency(**CE**): Peak number of 16-bit operations performed per second per mm^2.
- Energy Efficiency (**EE**): Peak number of 16-bit operations performed per second per Watt.
- Storage Efficiency (**SE**): Mega bytes of storage per mm^2. This includes synaptic storage in crossbars and neuron potential storage in SRAM central buffer.
- Energy consumed for entire state-of-the-art deep networks (VGG-NET and MSRA).

For our power and area analyses at 32nm technology, we use CACTI [90] for SRAM buffers, ORION 2.0 [58] for router and interconnect evaluation, the models of Shafiee et al. [111] for CMOS-compatible TaOx memristor crossbars, and recent adder/comparator models [84, 123]. We use [69] for ADC evaluation.

We use a manual process (emulating a future compiler) to map the different network layers of our workloads to crossbars/tiles while not exceeding any of the available resources, and while replicating layers to maintain a balanced pipeline. As described in Section 3.3.2, we estimate the length of a tick (107 ns) based on the worst-case $C + S$ value for our workloads. The length of each cycle is 0.78 ns and is determined by the latency to process one ADC sample. The overall performance is determined with an analytical model that considers the sizes of each network layer. For our evaluvation we calculate the $C + S$ value for each layer. The delay of one access to the central buffer dictates the frequency in configurations with really large central buffer. Note that cycle-accurate simulations are not required because the workloads do not encounter any conditional structural hazards.

Hunsberger et al. [51] show that convolutional neural networks can be mapped to SNNs while achieving very similar accuracy. In a similar vein, we use two state-of-the-art deep convolutional networks for image classification, VGG-NET [115] and MSRA [44], to evaluate INXS. Since we need to find a design point that performs well on both fully connected SNNs and convolutional SNNs, we pick VGG-NET (it has the most neurons/layer in Conv1) and MSRA (it has the most number of inputs/neuron in FC1).

ISAAC uses a 128×128 crossbar and an 8-bit ADC so bits are never dropped. The use of a modest crossbar size keeps noise in check and allows use of a low-resolution ADC. Given the inherent sparsity of spikes in an SNN, we allow use of a 256×128 crossbar while still using an 8-bit ADC. We also explore the use of a 6-bit ADC that assumes sufficient sparsity in spikes. Note that applications with high spike rates would be forced to use a subset of crossbar rows so the ADC precision is rarely exceeded. As a sensitivity study, we also explore use of a 6-bit ADC in tandem with 1-bit memristor cells that is guaranteed to not drop bits – while this design has a lower overhead for ADCs, it uses more crossbars to represent synaptic weights. Our results show that this design point does not match a design with an 8-bit ADC in tandem with 2-bit memristor cells, so we will not discuss it further.

3.5 Results

3.5.1 INXS Design Space Exploration

For all our results, we evaluate metrics across a number of design points. The X-axis in most figures describes these design points as $a \times b \times c \times d$, where $a \times b$ describes the number of crossbars in a Synaptic Unit, c represents the number of Synaptic Units per tile, and d represents the capacity of central buffer in a tile (in KB). Note that in a convolutional layer, a single crossbar can produce results for several neurons across several ticks. The size of the central buffer puts a cap on the number of neurons that can be produced locally by the corresponding synaptic units.

We first evaluate peak computational efficiency, shown in Figure 3.7, for the INXS design as we vary our design parameters. Similarly, Figures 3.8 and 3.9 quantify the EE and SE metrics respectively. The main observation from these figures is that all these metrics improve when the central buffer size is reduced. This is because peak metrics are primarily impacted by the number of crossbars, which offer high storage and computation. Note that SE is a sum of neuron and synaptic density. Providing a large SRAM buffer increases neuron density (and is helpful to convolutional layers), but decreases synaptic density (not helpful to fully connected layers). Clearly, the latter effect is more dominant in this analysis of peak performance, so we see a drop in SE when CB size is increased. Figure 3.10 further breaks the SE metric into neuron and synaptic density.

While peak CE, EE, and SE are useful metrics and favor crossbar computation over neuron potential storage, deep networks with large convolutional layers benefit more from neuron potential storage. Therefore, ultimately, we need to evaluate INXS designs on state-of-the-art deep networks, e.g., VGG and MSRA. Figures 3.11 and 3.12 show the energy for different design points for these two workloads. These real workloads exhibit the best metrics when using larger central buffer sizes. Based on this analysis, we pick an ideal design point that does reasonably well for both workloads: 8x8x8x128. For this ideal design point, Figures 3.13 and 3.14 show a breakdown of the area and energy required by the different layers of the deep networks. Table 3.2 summarizes the area and power of each component in an INXS tile.

Contrary to what we saw earlier for peak CE, EE, and SE metrics, the configurations with large central buffer perform well on MSRA and VGG-NET. The reason for this is

the overhead of inter-tile and intra-tile interconnect energy. Configurations with small central buffers engage in more intra- and inter-tile communication which increases energy significantly. This is especially true for convolutional layers. Fully-connected classifier layers, on the other hand, benefit more from crossbars than from large buffers. But in these large workloads, the convolutional layers dominate (see per layer breakdown in Figures 3.13 and 3.14).

3.5.2 Comparison to TrueNorth

Next, we compare INXS metrics to those of TrueNorth. Table 3.3 summarizes the key metrics. Because of the high parallelism in INXS, it achieves three orders of magnitude higher performance/area than TrueNorth. Because of the high ADC overhead, INXS has only a 10× improvement over TrueNorth in terms of EE. Using memristor xbars gives us significant density for synaptic storage compared to the SRAM storage used in TrueNorth. This also helps us eliminate the need for quantization of synaptic weights. INXS has a much larger advantage in terms of neuron density – it balances resources appropriately by recognizing that convolutional layers need large central buffers and low synaptic storage. The routing table employed in each neuron unit removes the severe constraints imposed by TrueNorth on the number of inputs/outputs to a neuron. As explained earlier, we achieve a more flexible architecture with the maximum number of inputs a neuron can receive being 512× higher than that TrueNorth.

Table 3.1. Comparison of accuracy, throughput, and energy efficiency for state-of-the-art ANNs and SNNs. The digital SNN numbers correspond to TrueNorth [4]. The energy number for the Analog SNN accelerator is for a small-scale 32-neuron implementation [78].

Type	Accuracy (%)			TOPs/s		pJ/op	
	MNIST	CIFAR-10	AlexNet	Digital	Analog	Digital	Analog
ANN	99.7 [19]	96.5 [40]	89 [68]	9 [5, 16]	45 [111]	3.2 [5, 16]	1.5 [111]
SNN	99.4 [31]	89.3 [32]	82.5 [50]	0.058 [4]	0.07 [78]	41 [4]	0.35 [78]

Table 3.2. INXS area and power breakdown (for one 8x8x8x128 tile configuration). SU: Synaptic Unit, NU: Neuron Unit.

Component	Description	Area (μm^2)	Power (mW)
Memristor xbar	512	8K	307.2
ADC	512	136K	1024
Shift and Add	512	30.7K	25.6
Router	128b Flit, 5-ports	253K	107
Input Buffer	4KB	489K	102.4
Output Buffer	2.5KB	36.6K	7.2
Central Buffer	128KB	1,738K	304
Interconnects	128b output bus, 128b ring bus	61K	135.7
Functional Units	8 adders and comparators	15K	4
Routing table	12.75KB	21K	12.1
Tile	64 xbars/SU, 8 SU, 8 NU	**2.8** mm^2	**2030 mW**

Table 3.3. INXS comparison with TrueNorth. ND-Neuron density, SD-Synaptic density

Metric	TrueNorth	INXS	Improvement
EE (GOPS/s/W)	400	4.1K	10.4x
CE (GOPS/mm^2)	0.703	2.2K	3129x
SE (MB)/mm^2	0.138	2.06	15x
ND (Neurons/mm^2)	2.73K	866K	363.5x
SD (Synapses/mm^2)	0.65M	1.497M	2.2x
Neuron connectivity	256	128K	512x

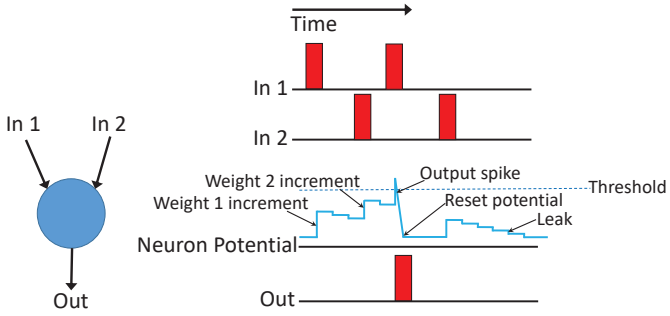

Figure 3.1. A basic 2-input LLIF spiking neuron. The figure shows how the neuron potential is incremented when input spikes are received, how a leak is subtracted when there are no input spikes, and how an output spike is produced when the potential crosses the threshold.

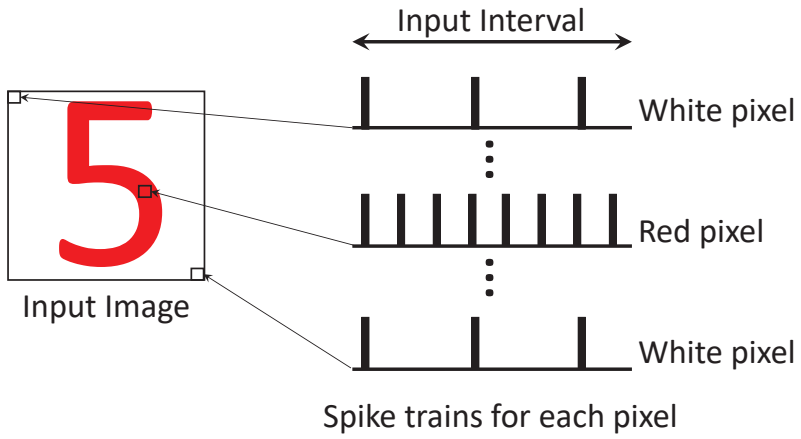

Figure 3.2. Example of an input image being converted into a number of input spike trains that are fed to a spiking neural network.

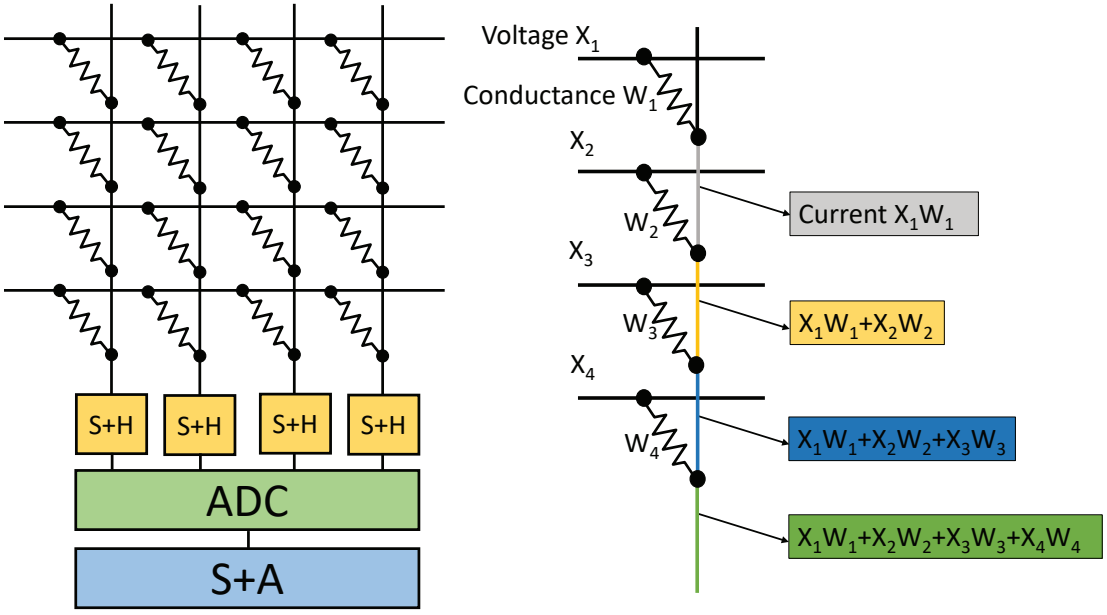

Figure 3.3. In-situ computation using memristor xbar. (a) An example 4x4 memristor xbar connected to peripheral circuits. (b) The conductance of the memristor corresponds to the synaptic weight and the voltage corresponds to the spike input. The current at the end of the bitline is the dot-product of input spikes and synaptic weights.

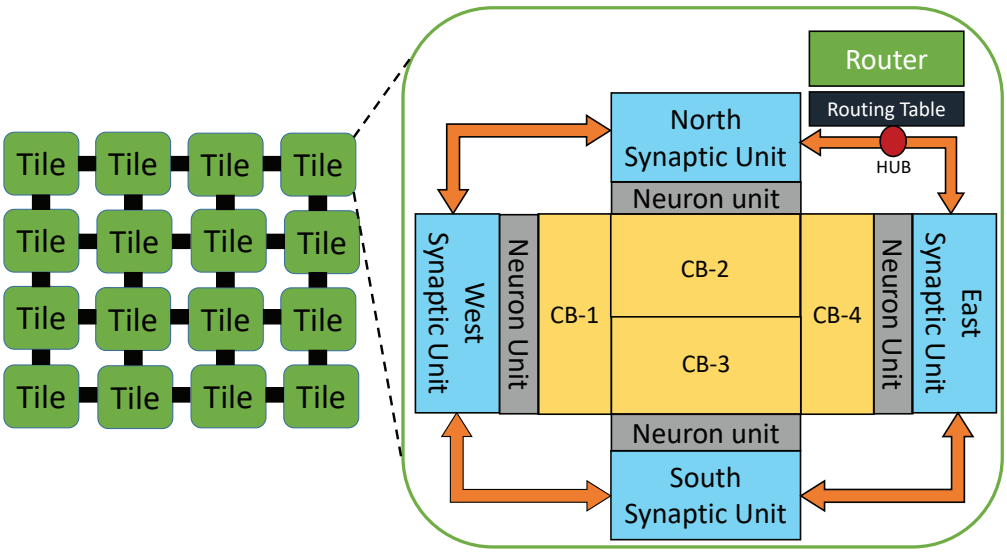

Figure 3.4. INXS tiled architecture and details of one tile.

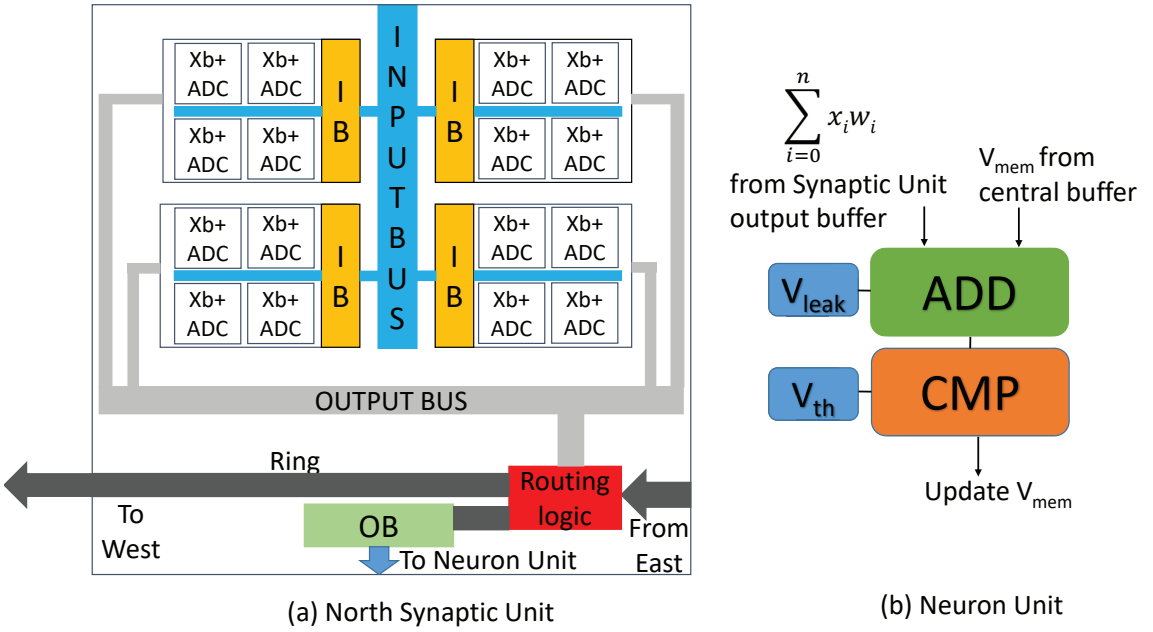

Figure 3.5. INXS architecture. (a) Synaptic Unit. (b) Neuron Unit. (The exact number of xbars, ADCs, adders, etc. vary in our optimal design points.)

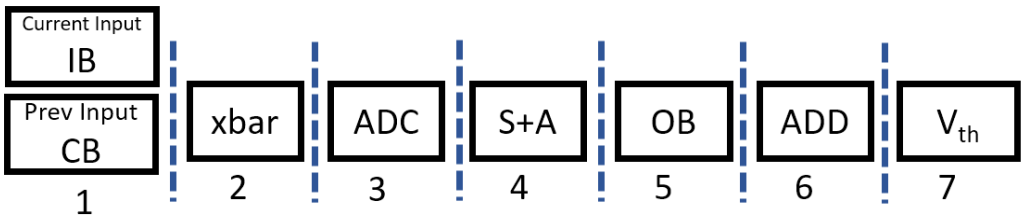

Figure 3.6. INXS pipeline. IB - Input Buffer, CB - Central Buffer, OB - Output Buffer

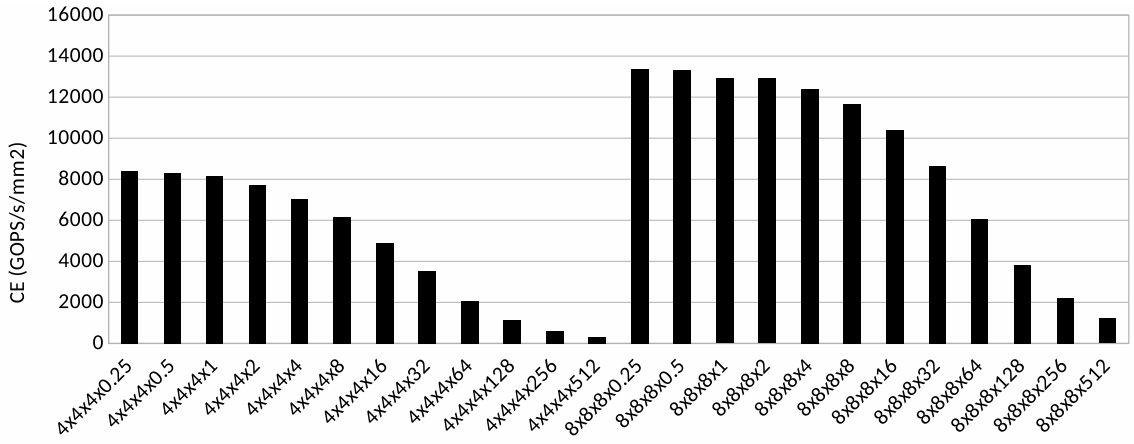

Figure 3.7. Computational efficiency of INXS for various configurations.

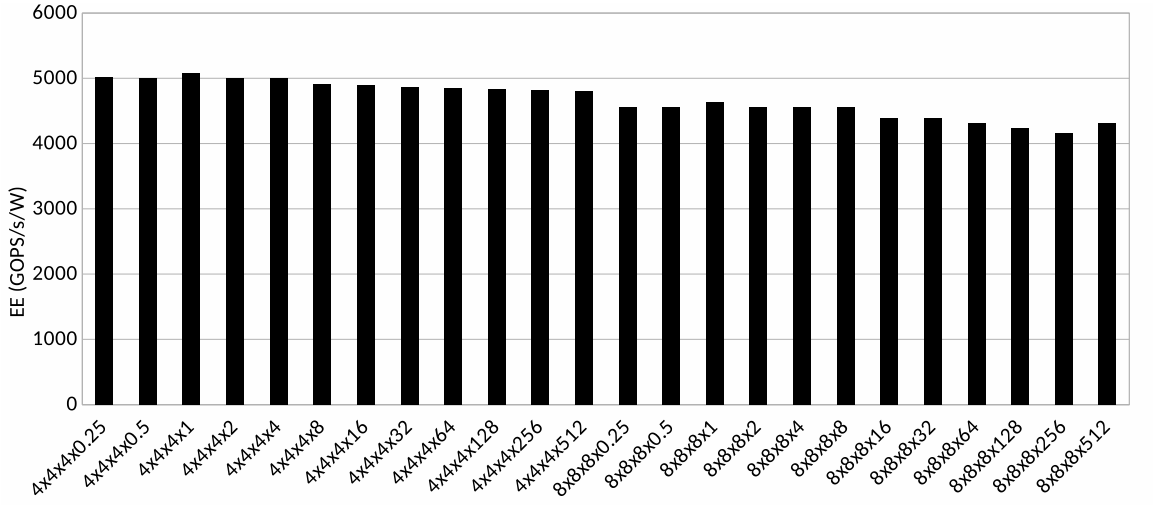

Figure 3.8. Energy efficiency of INXS for various configurations.

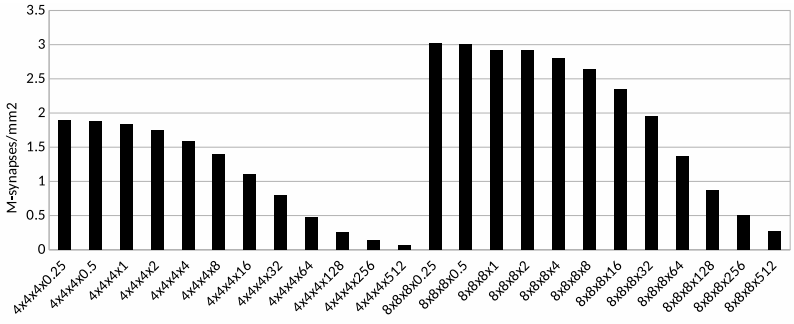

Figure 3.9. Storage efficiency of INXS for various configurations.

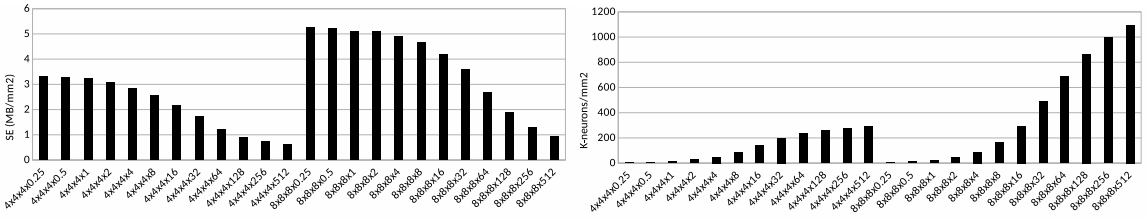

Figure 3.10. M-synapses/mm^2 and K-Neurons/mm^2 for various INXS configurations.

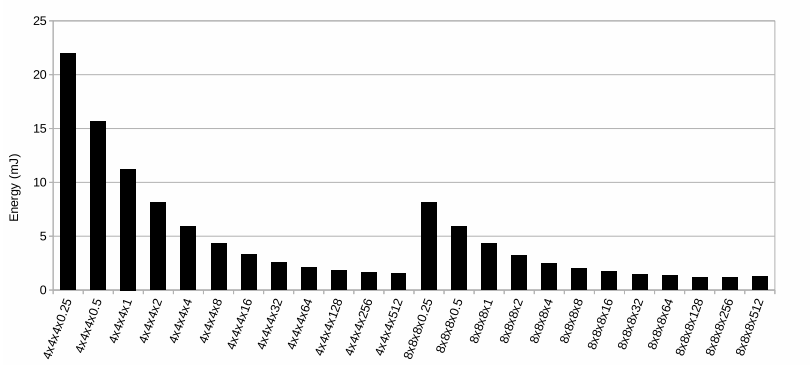

Figure 3.11. Energy (for 1 image) estimates of different INXS configurations for VGG-NET.

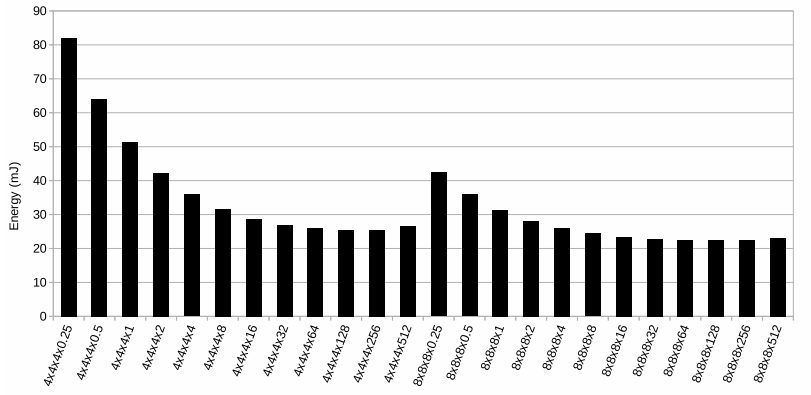

Figure 3.12. Energy (for 1 image) estimates of different INXS configurations for MSRA.

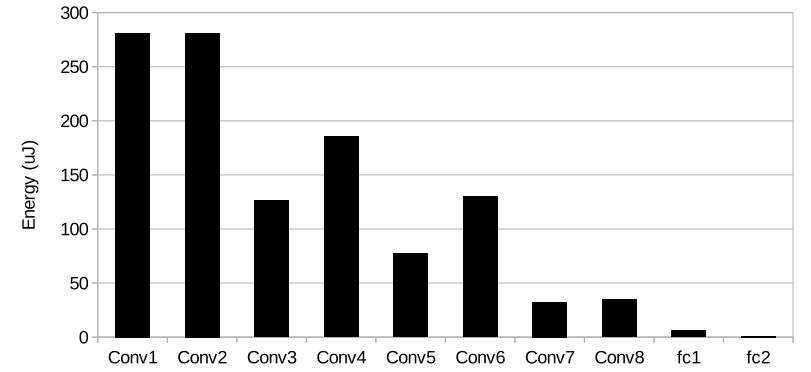

Figure 3.13. Energy estimate of INXS (8x8x8x128) for different layers of VGG-NET.

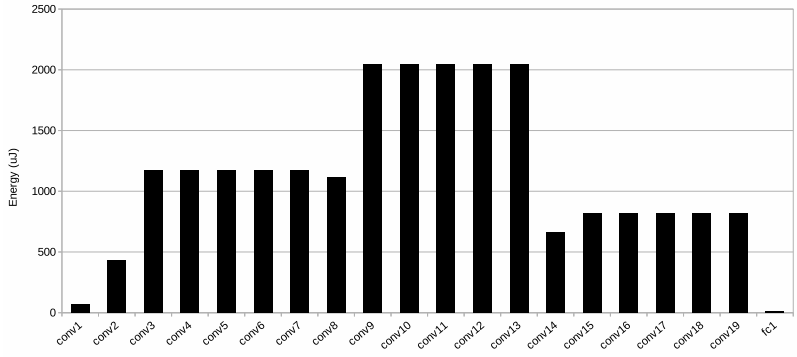

Figure 3.14. Energy estimate of INXS (8x8x8x128) for different layers of MSRA.

CHAPTER 4

SPINALFLOW: AN ARCHITECTURE AND DATAFLOW TAILORED FOR SPIKING NEURAL NETWORKS

4.1 Introduction

Inspired by Neuroscience, researchers have explored the potential of Spiking Neural Networks (SNNs) to achieve high prediction accuracies for various image and speech applications [4, 8, 30, 32, 116]. A spiking neuron is stateful; it maintains a potential based on previously seen inputs; as binary input spikes are received, the potential is moved up or down; a binary output spike is produced when the potential reaches a threshold. Spiking neurons mimic the operations in biological neurons, in hopes that emulating the brain will provide very high prediction accuracy at very low energy [116]. However, silicon implementations of SNNs have generally lagged behind state-of-the-art silicon implementations of ANNs. In spite of this, SNN advancements are important because of their potential benefits in specific applications. For example, a Google project shows a small-scale SNN with sparse temporal codes achieving higher accuracy than a similar-sized ANN using higher precision [20]. In the short term, SNNs are expected to be effective and useful in the following scenarios: (i) when a large/labeled training set is not available, (ii) when the inputs are expected to deviate from the training set, (iii) when continual learning [17, 92] is necessary, and (iv) to establish initial neural network weights before engaging resource-intensive training approaches [72]. In the long term, researchers need to build on the work of Comsa et al. [20] and develop new training techniques to further exploit the information content in relative spike times so that SNNs can be competitive with the best ANNs under all circumstances. The future potential of SNNs is also echoed by the many commercial projects on SNN hardware – IBM TrueNorth [4], Qualcomm Zeroth [101], Intel Loihi [24].

As Smith lays out in his FCRC'19 keynote [116], much work remains in developing

SNN training methods and architectures. In theory, SNNs naturally exhibit high sparsity, i.e., they can pack the information content of an 8-bit ANN into the relative timing in a sparse spike train within a modest time window. This work attempts to realize the low energy potential of SNNs while overcoming the temporal dimension.

Unfortunately, modern SNN architectures achieve lower throughputs and higher energy per neuron, compared to ANN accelerators [4, 24, 28, 61]. This is primarily because of the temporal aspect in SNNs – inputs are received and processed across multiple ticks. Not only does this require more time, it also puts constraints on architecture dataflows and the data reuse that can be exploited. Further, the dataflow must also manage reuse for a large data structure unique to SNNs: neuron potential.

We therefore create a baseline SNN architecture, Spiking-Eyeriss, that is modeled after a canonical ANN accelerator Eyeriss. We observe that processing SNNs (both rate-coded, and temporal-coded) in the traditional way imposes significant data movement for neuron potentials in every tick. This problem is exacerbated by the Eyeriss row-stationary dataflow, in which partial-sums (or neuron potentials) are not fully accumulated before being offloaded to its global buffer. To address this problem, we introduce a new accelerator, SpinalFlow, that processes spikes in a compressed, and chronologically sorted manner in a single time-step (like ANNs). Using row-stationary dataflow for this approach would lead to non-trivial sorting overheads; this is addressed by adapting the dataflow to use an output-stationary model. The proposed dataflow does require repeated accesses of weights from a large buffer. This large buffer reduces the compute density of SpinalFlow, relative to the ANN baseline, for some workloads. This drawback is alleviated when dealing with low resolution inputs and sparse spike trains, which is an inherent property of temporally coded SNNs. The architecture is thus designed to naturally exploit the expected high sparsity in SNN spike trains [88, 109]. Relative to the baseline ANN, SpinalFlow has simpler processing elements, but an additional hardware merge-sort unit and a large buffer to exploit weight reuse.

The main contributions of this chapter are:

- An analysis of the inefficiencies in a baseline SNN design.

- A representation for spike inputs and outputs that is compressed, time-stamped, and

sorted.

- An SNN architecture and dataflow that is tailored for this input/output representation and that increases reuse of neuron potential, input spikes, and weights.

- A 1.8× average energy improvement at 4-b resolution and 90% sparsity over a 4-bit version of Eyeriss, a 5× average energy improvement at 4-b resolution and 5.4× average latency improvement at a sparsity of 90% over 8-bit Eyeriss, a 1.16× average energy improvement at a sparsity of 90% over an SCNN [98] baseline, and an order of magnitude energy improvement over the baseline SNN architecture.

- The chapter thus shows that SNN architectures can complete the computations required for inference in similar time and energy as an ANN architecture. This can significantly impact platforms, e.g., those requiring real-time learning, where SNNs have the potential to achieve higher accuracies than ANNs.

4.2 Background

4.2.1 Spiking Neurons

A number of projects [4, 8, 62] have attempted to implement biologically plausible neuron models in hardware. Many of these hardware projects implement neuron models that are highly simplified, but that can emulate many biologically observed neuron behaviors, e.g., the Izhikevich neurons [94]. The most popular of these simple neuron models is the Integrate and Fire model (IF), shown in Figure 4.1. An IF neuron is stateful – it retains the value of its (membrane) potential. This *neuron potential* reflects inputs that have been received in the recent past. Inputs are received in the form of binary spikes. When a spike is received on an input, the synaptic weight for that input is added to the potential (see Figure 4.1). In every tick, a leak is also subtracted from the potential. When the neuron's potential eventually reaches a specified threshold, the neuron produces an output spike of its own. After the spike, the neuron potential is reset.

Spiking neurons have the potential to be hardware-efficient because inputs and outputs are binary spikes. The spiking neuron model does not require a multiplier – because the input is binary, the synaptic weight is simply added to the potential. Spikes therefore can

lead to efficient communication *and* computation. This is a key feature of SNNs, but as we show later, modern SNNs have failed to exploit this advantage.

Because a neuron is designed to respond after observing spikes over time, an input (say, an image) is provided over an *input interval*, e.g., 16 ticks[1]. Figure 4.2 shows how each pixel of an input image is converted into a spike train that extends across an input interval. These spike trains are fed as inputs to the first layer of neurons. Prior work has primarily used *rate codes*, where for example, a red pixel value may be converted into 8 spikes in 16 ticks, while a blue pixel value may be converted into 12 spikes in the input interval. A *temporal code* converts an input pixel value into a single spike at a specific time, e.g., a red pixel value results in a single spike in the 8th tick, while a blue pixel value results in a single spike in the 12th tick. The code also includes stochasticity, e.g., a rate code may use a Poisson distribution to inject spikes [3].

We refer to rate-coded and temporally-coded SNNs as *r-SNN* and *t-SNN* respectively. Since biological neurons work at low resolution and because t-SNNs work better at low resolution, t-SNNs use short input intervals [63, 88, 116]. t-SNNs also exhibit high levels of sparsity, i.e., under 10% of all neurons produce a spike in an input interval [63, 109].

Spiking neurons are typically trained with a biologically plausible process called STDP (Spike Timing Dependent Plasticity [34]). This is an unsupervised training method where each neuron adjusts its weights based on a local process to estimate a spike's relevance [28, 110]. To increase accuracy, some recent works have also resorted to supervised backpropagation-based training for SNNs [31, 32]. The recent work of Comsa et al. [20] also employs this approach to train a t-SNN to achieve the same accuracy as an ANN.

This is a key point. *A t-SNN with its inherent lower resolution, higher sparsity, and ability to find correlations in inputs can match the accuracy of an ANN with higher-resolution operands. But the efficiency advantages of t-SNNs will not be evident until we improve its dataflow and operand reuse.*

[1]A *tick* is the minimum unit of time in an SNN. In one tick, a neuron evaluates its inputs, updates its potential, compares against its threshold, and produces a spike if necessary.

4.2.2 SNN Accelerators

A variety of digital and analog SNN accelerators have been described in the literature [4, 24, 37, 62, 77, 78, 91, 124]. IBM's TrueNorth processor [4] is the most prominent example of a digital architecture for SNNs. TrueNorth is composed of many tiles, where each tile implements 256 neurons, each with 256 inputs. In every 1ms tick, a tile processes all received input spikes and sends any resulting output spikes to neurons in the next layer. Within a tick, the tile sequentially walks through every neuron in that tile and every input spike to perform several updates to each neuron potential. TrueNorth achieves relatively poor throughput and latency because of its 1 ms tick. It also does not have any parallelism within a tile. To enable an apples-to-apples comparison with state-of-the-art ANNs, we design new baseline SNN architectures that borrow some of the ANN accelerator best practices. This baseline is described in Section 4.3.1 and offers orders of magnitude better throughput and latency than TrueNorth.

4.2.3 ANN Accelerators

It is worth noting that in contrast to spiking neurons, artificial neurons rely on dot-product calculations, they do not retain state across consecutive inputs, and they are typically trained with supervised back-propagation based stochastic gradient descent. A number of ANN accelerator designs have been proposed in recent years [14, 18, 64, 111, 112]. In this work, we use Eyeriss [15] as a baseline because it captures many key innovations, it has been implemented in silicon, and it has many publicly available details/tools. Eyeriss uses a hierarchy of global buffers and scratchpads/registers scattered across a grid of processing elements (PEs). It uses a row-stationary dataflow for its PEs. Each PE processes a row of computation for some kernels and some input feature maps, thus exploiting reuse. The partial sums, feature maps, and kernels then move to an adjacent PE to continue the computations with high reuse. Such dataflows are a key feature in most state-of-the-art ANN accelerators, e.g., the Google TPU [56]. Many ANN accelerators also leverage sparsity in weights and/or activations [98, 112, 141]. To save energy, a multiplier/adder in Eyeriss skips its operation if either operand is zero. Architectures like SCNN [98] can also exploit ANN sparsity to reduce execution time.

4.3 Understanding Sources of SNN Inefficiency

4.3.1 Defining the ANN and SNN Baselines

4.3.1.1 ANN Baseline

To identify sources of SNN inefficiency, we use the Eyeriss architecture [15] as an ANN baseline. Eyeriss has the basic optimizations (dataflow, reuse, ineffectuals) that are widely adopted in both academic and commercial accelerators [56]. Much of our analysis focuses on an 8-bit version of Eyeriss, while the SNN models employ a 16-tick input interval and temporal codes with high sparsity. While the SNN design has higher sparsity and lower resolution, those advantages are inherent in the SNN design, i.e., we are not artificially introducing an accuracy/efficiency trade-off. To further understand the relative merits, we also compare SNNs (with varying resolutions) to an ANN that engages the accuracy/efficiency trade-off and operates at low resolution. Note that a 4-bit Eyeriss has the same input resolution as a 16-tick SNN.

Eyeriss has a grid of processing elements (PEs) that are fed with inputs/weights and partial sums from a global buffer. Each PE has a MAC unit and a scratchpad that stores a row of an input feature map (ifmap), a row of a filter, and partial sums (psums) for the output feature map (ofmap). Figure 4.3a shows the components in a single PE at 8-bit resolution. A PE can elide a computation to save energy if either input is zero. Within the 2D grid of PEs, ifmaps are shared diagonally, filters are shared horizontally, and psums are accumulated vertically. This is referred to as *Row-Stationary* dataflow [15].

4.3.1.2 SNN Baseline

Our SNN baseline, *Spiking-Eyeriss*, closely follows the Eyeriss architecture. The resulting PE, shown in Figure 4.3b and summarized in Table 4.1, does not have a multiplier unit. Ifmap scratchpads only have a width of 1 bit. Weights and partial sums have the same width as the ANN baseline. After comparing the potential to the threshold, a 1-bit neuron output is produced; this 1-bit neuron output and the 8-bit neuron potential must both be saved in the global buffer or in off-chip DRAM. The 8-bit neuron potentials have to be retained for the entire input interval.

The PE grid in the SNN is more efficient than in the ANN (no multipliers, ifmap scratchpads are only 1-bit wide). Even though the inputs to a layer have shrunk in size, the overall

memory requirements in the SNN are higher because every neuron's potential must be retained. The architecture is agnostic to the type of data encoding used (rate or temporal). As described shortly, this SNN baseline offers significantly higher throughput/area and throughput/power than the state-of-the-art SNN architecture, TrueNorth [4].

In our analysis, we employ *Tick Batching*, where the PEs work on all ticks of the input interval for a layer before moving on to the next layer. In tick batching, the reuse distance for membrane potential is short and with appropriate tiling, the membrane potentials can be primarily accessed out of the global buffer. Meanwhile, the output spike train from a layer is likely too large to fit in the global buffer and may have to be saved in off-chip DRAM. We have analyzed other forms of batching, e.g., processing all layers before examining the next tick, and concluded that tick batching leads to overall lower data movement.

4.3.2 Evaluation Parameters

To evaluate Eyeriss and Spiking-Eyeriss, we developed an energy/performance model that captures the latencies, energy, and throughput for different networks. The model takes in the layer specifications as input, and outputs the number of accesses to different registers, scratchpads, buffers, and off-chip memory. Based on these statistics, and the average energy per operation, we calculate the energy per layer. The model uses analytical equations to capture the dataflow of Eyeriss (and Spiking-Eyeriss) based on layer dimensions and how they map to the PE array. This mapping and resource contention ultimately dictate PE utilization and performance. Note that our analysis models the zero-gating technique of Eyeriss where filter scratchpad and ALUs are gated when a zero-valued input activation is encountered. Table 4.2 summarizes the evaluated networks and the input image sizes. We consider a range of synthetic single-layer networks to understand how the architectures impact each type of network topology. We also consider three full-fledged networks [47, 63] (ResNet, MobileNet, and STDP-Net) that incorporate a number of varied layers. Previous research [36, 49] has shown that the visual cortex is organized as a cascade of simple and complex cells structured similar to a CNN.

We implement and synthesize the processing elements of both Eyeriss and Spiking-Eyeriss using 28 nm FDSOI technology node. First, we modeled the behavior of the PEs in

verilog and synthesized it using Synopsys Design Compiler. We then used Innovus for the backend flow in order to get accurate post layout metrics (area, delay and power) by taking the parasitics into account. To model the global buffer, we use CACTI [90] and scale the output from 32 nm to 28 nm technology. To reduce the number of variables in our study, we do not attempt memory compression, but simply assume a low-energy HBM-like memory interface at 4 pJ/bit [95]. Based on the above methodology, we calculate an average energy per operation for all components. This is combined with the number of operations for each component to generate the overall energy consumption for each layer. Table 4.1 has details on each accelerator component. The primary metrics, while keeping area roughly the same, are: energy per inference, and latency per inference.

4.3.3 Analysis of Spiking Eyeriss

We first try to estimate the efficiency gap between our baseline SNN and ANN. We consider the impact of rate coding (r-SNN), temporal coding (t-SNN), sparsity, and resolution on SNN energy. Figures 4.4 and 4.5 show the energy consumed by r-SNN and t-SNN respectively on Spiking-Eyeriss. All data points are normalized against an ANN operating at resolution of 8 bits and its typical activation sparsity of 60%, i.e., 60% of activations are zero.

At high resolution and low sparsity, Spiking-Eyeriss consumes an order of magnitude more energy than Eyeriss. This is because of the need to repeatedly update neuron potential and fetch a weight multiple times across the input interval.

For sparse inputs, Spiking-Eyeriss and Eyeriss can avoid filter reads and partial-sum updates. Because t-SNNs encode non-zero inputs with a single spike, they consume less energy than r-SNNs. The difference is 1.63×, 1.03×, 1.08×, and 1.004× at 8bSp60, 8bSp98, 2bSp60, and 2bSp98 respectively, i.e., as one might expect, the gap shrinks when spike activity is low. More noteworthy is the "crossover point", when the SNN energy falls below that of the baseline Eyeriss. For r-SNN and t-SNN, this happens at 3bSp90 and 3bSp60 respectively. This quantifies the sparsity and resolution required for an SNN to overcome its inherent disadvantage of managing neuron potentials across many ticks and spikes.

4.3.4 Summary

The Spiking Eyeriss baseline has a peak throughput/area of 70 $GOps/s/mm^2$, which is 519× higher than that of TrueNorth. Truenorth is a design optimized for high levels of spike sparsity, and therefore low leakage and low energy per neuron, so its peak throughput/watt is comparable to that of Spiking Eyeriss with temporal codes and 90% sparsity. In spite of this throughput advancement relative to TrueNorth, there is a wide gap between the ANN and SNN baselines. *The performance gap (16×) is because of the need to process all (16) ticks in the input interval.* In terms of energy, we see that temporal coding is clearly superior. While a large fraction of prior SNN work has focused on rate coding, we note here that algorithmic advances in temporal coding are required so that its inherent energy efficiency can be leveraged. For the rest of the study, we will focus on the t-SNN with tick batching. However, *this SNN baseline with 90% sparsity and 16-tick intervals consumes 2× more energy per inference than the ANN because of its repeated accesses to neuron potential and filter weights*. We next devise techniques to shrink this gap.

4.4 Proposed SNN Architecture: SpinalFlow

Our analysis has shown that the key drawback in our baseline SNN is the need to iteratively process (say) 16 ticks for every input interval, which in turn takes more time and requires many accesses to the memory hierarchy to update neuron potential and read filter weights. While the row-stationary dataflow of Eyeriss is highly efficient for ANNs, we must devise a new dataflow that caters to the temporal aspects and new reuse patterns exhibited by SNNs. We refer to this new architecture as *SpinalFlow*.

4.4.1 Terminology

Before we start, Figure 4.6 is a quick summary of the terms we'll use in our description. We first discuss the operations in a CONV layer. Figure 4.6 shows a layer with C ifmaps each of dimension H×W. Each of the K R×S×C kernels is convolved with the entire ifmap to produce K ofmaps. When the first kernel is applied to the first receptive field, i.e., the first R×S×C grid in the ifmap, the first neuron in the first ofmap is produced (the darker region in the 1st ofmap). Likewise, when K kernels are applied to this same grid of inputs, the 1st neuron in all the ofmaps are generated as shown by the dark region in Figure 4.6.

We refer to these K neurons as a *spine*. In our discussions, we assume the value of K to be 128. The computation is ordered such that we produce the first spine of the ofmaps, followed by a shift of the input receptive field to produce the second spine, and so on.

4.4.2 Hardware Organization

We use an array of 128 PEs. Each PE has an accumulator (register and adder) and a comparator. Each PE is responsible for producing a neuron in an ofmap spine. Figure 4.7 shows the first step, where the PEs are responsible for the first spine of the ofmaps; PE-1 produces the first neuron for ofmap-1, PE-2 produces the first neuron for ofmap-2, and so on. Similar to our baselines, these PEs are fed by a global buffer that stores kernel weights. The PEs use a form of output stationary dataflow, i.e., PE-1 is dedicated to work on the first neuron for ofmap-1 till all its inputs are processed. Over the next many cycles, the PEs will receive all the spikes in their input receptive field for their input interval. The *Input Buffer* provides these input spikes. The example in Figure 4.7 shows that the input spikes are chronologically sorted: $< 1, 17 >, < 2, 1926 >, < 3, 75 >, < 3, 460 > \ldots$, i.e., input 17 has a spike in tick-1, input 1926 has a spike in tick-2, input 75 has a spike in tick-3. Note that in our example, the receptive field has a size of 2K, and every neuron in that receptive field can only spike at most once in its input interval (temporal code), so the input buffer can have up to 2K entries. 128 PEs also require that the global buffer feed 128 different weight values, therefore demanding a wider bus than in the baseline. We later factor this in our evaluation.

- **Step 1: Cycle 1.** Step 1 produces the first spine in the ofmaps; producing this first spine can take up to 2K cycles. In the first cycle (shown in Figure 4.7), we examine the first entry in the input buffer. It represents a spike in input 17 in tick 1. Each of the PEs' neuron potential must be incremented by their corresponding kernel weight. We therefore read a row of 128 weights from the global buffer, corresponding to the 17th entry of all 128 kernels. Each PE receives one of these 128 weights and the weight is added to the neuron potential. The neuron potential is compared to its threshold. In our example, PE-110 has exceeded its threshold in tick-1, so it produces a spike $< 1, 110 >$ that is placed in its output buffer. After producing its spike, PE-110 idles for the rest of the input interval because a neuron can only produce one spike

in its input interval.

- **Step 1: Cycle 2.** In the next cycle, the next spike in the input buffer is processed (shown in Figure 4.8). This happens to be input 1926 spiking in tick 2. The row of 128 weights corresponding to input 1926 are read from the global buffer and fed to the PEs. Another set of neuron potential increments is performed at the PEs. PE-73 produces a spike and idles. The output buffer is appended with this new spike at tick-2: $< 2, 73 >$. We see that the spikes in the output buffer are naturally sorted, i.e., the first spine of the ofmaps is represented as a list of chronologically sorted spikes.
- **End of Step 1.** The process repeats for up to 2K cycles until all spikes in the input buffer have been processed. Since some of the neurons in the previous layer may not have spiked in their input interval, the actual processing time can be variable and much less than the worst-case 2K cycles. In our evaluation, we assume the worst case, and leave the exploitation of activation sparsity for future work. The output buffer now contains up to 128 chronologically sorted output spikes, corresponding to a spine of the ofmaps. In practice, each spine will have less than 128 entries (since several neurons may never spike in an input interval). This spine is then written into the global buffer (see Figure 4.9) and will be used later as input to the next convolutional layer.
- **Starting Step 2.** We are now ready to move to step 2, where the PEs are responsible for computing the 2nd spine of their ofmaps. The PEs reset their neuron potentials to zero. Before we start step 2, we must shift the receptive field and create a new input buffer with sorted spikes within the new receptive field. Note that the previous layer produced sorted spines of its ofmaps, which now serve as sorted ifmap spines for the current layer. To create the sorted receptive field, we must first read 16 of these ifmap spines from the global buffer into 16 ifmap spine buffers. As shown in Figure 4.9, these 16 pre-sorted 128-entry spines can be merge-sorted to produce the sorted 2K entries that represent the input receptive field. The 16 ifmap spine buffers and the merge-sort unit have replaced the conceptual sorted input buffer that we showed in earlier figures. The merge-sort unit is simply a tree of comparators that, in every cycle, picks the smallest entry among the heads of each ifmap spine buffer. Depending on the spine that produced that entry, an offset is added so the correct

row of weights is accessed. To initiate every step, 16 cycles are required to populate the ifmap spine buffers. This is a small overhead for convolutions since the step requires hundreds of cycles of computations.

4.4.3 Summary

With this spine-oriented output-stationary dataflow, the proposed *SpinalFlow* architecture no longer suffers from the drawbacks in our baselines. Because we are using temporal codes and because we sequentially walk through time-stamped spikes, we need exactly as many computations as the ANN baseline, i.e., we are no longer penalized by the multi-tick input interval. Creating the compact sorted list of time-stamped spikes is trivial because of how spikes are produced by the previous layer. The architecture can yield speedups with activation sparsity with zero change, while for a similar performance effect, an ANN requires more invasive changes [5, 98]. By using an output stationary dataflow, a neuron is mapped to its PE for the entire input interval. The PE initializes its neuron potential accumulator to zero, increments it as spikes are received, produces a spike when the threshold is crossed, and discards the neuron potential before moving on to the next neuron. We are thus eliminating separate storage and data movement for neuron potential. Note that our dataflow focuses on maximizing neuron potential reuse and parallelism across a spine because of the need to sequentially process each tick; Eyeriss on the other hand optimizes a combination of reuse of inputs, kernels, partial sums.

4.4.4 Hardware Details

We observed that provisioning SpinalFlow with as many resources as Eyeriss led to a sub-optimal design. We therefore provide as many resources as required for the common case observed in our dataflow.

We use 128 PEs in our design because the number of feature maps per layer in large convolutional networks is often a multiple of 128. The overall architecture of SpinalFlow is shown in Figure 4.10a, and the details of one PE are shown in Figure 4.10b. The pseudo-code for our dataflow is shown in Figure 4.10c. Each PE in our design is much simpler than the PE in Eyeriss. Since we are no longer processing an entire row at a time (the row-stationary dataflow of Eyeriss), the PE does not require large scratchpads. Such scratchpads occupy half the core area in Eyeriss, so this is a significant saving.

The SpinalFlow global buffer from earlier figures is split into a *Filter Buffer* and *Input Buffer*. While Eyeriss retains most of its weights in scratchpads, SpinalFlow retains its weights in a Filter Buffer. This buffer has to store all the weights in a receptive field for several filters because any of those weights may be required in a step. To accommodate some of the large convolutions in our workloads, we employ a 576 KB Filter Buffer organized into 32 banks, with a row width of 128 bytes (providing 128 1-byte weights at a time). In order to feed weights to 128 PEs in a cycle, the Filter Buffer needs an output bus width of 1024 bits. Depending on the type of layer being executed, bank assignment for weights and feature maps can be configured.

The inputs are received from a 9 KB Input Buffer, which stores the neuron ids and spike times for multiple spines. The spines for the input receptive field are fed to a *Min Finder* circuit (a tree of comparators) that identifies the next chronological spike and uses that neuron id to read a row of weights from the Filter Buffer. The PE array output is marshalled into an output queue that is eventually written to off-chip memory.

To evaluate the power and area of the processing elements and Min Finder, we adopt the same synthesis and SPICE methodology described in Section 4.3.2. To model the Input and Filter SRAM buffers, we use CACTI [90]. Area and power estimates are summarized in Table 4.3. We do not add the other exotic SNN features that can be found in TrueNorth (leak, stochasticity, various operation modes). We leave this for future work and note that the PEs account for a small fraction of chip area. Note that SpinalFlow seamlessly handles sparsity, which is an important feature in SNNs, i.e., a neuron that doesn't spike does not consume any resource bandwidth.

4.4.5 Other Networks

For small networks, where an input spike is seen by fewer than 128 neurons in the next layer, the PEs will be under-utilized. This is the uncommon case in our workloads. For larger networks, the computation has to be decomposed to work on 128 output feature maps at a time. The filter buffer has been sized large enough to accommodate all weights for 128 filters in our large convolutional layers. Once the filter buffer is loaded, it is reused several times to completely process the corresponding output feature maps.

The demands of a fully-connected network are different. Typically, the input receptive

field is large. The spikes in this receptive field have to be chronologically sorted beforehand, with potentially multiple hierarchical passes over the MinFinder circuit (which can only handle 16 128-entry spines at a time). The sorted list is then reused for a large set of output neurons. At a time, the PEs can process 128 output neurons. The entire input spike train for these 128 output neurons is processed before we move to the next 128 output neurons. Similar to most accelerators like Eyeriss and TPU, a fully-connected layer exhibits no weight reuse and is typically limited by the memory bandwidth required to fetch weights. Based on the input spike, a set of weights is fetched from memory, fed to the PEs, and then discarded. The only way to improve weight reuse and PE utilization is with batching, e.g., process 100 images at a time. In an SNN, such image batching is only effective if all the weights for the layer can be retained on the chip at a time (since each image in the batch has to fetch weights corresponding to its next input spike). We evaluate this in the next section.

4.5 Results

4.5.1 SpinalFlow versus Eyeriss

4.5.1.1 Energy Comparison

Figure 4.11 shows the energy per inference of SpinalFlow for synthetic conv layers normalized to Eyeriss. The early analysis assumes 8-b resolution and 60% activation sparsity for Eyeriss; we later also consider lower-resolution versions of Eyeriss. Along the X-axis, we vary SNN sparsity and resolution for SNN activations and weights. Even at 8b resolution and 60% sparsity, for most synthetic workloads, SpinalFlow consumes less energy than Eyeriss. This is mainly because of the way SpinalFlow handles sparsity. Figure 4.12 shows the energy breakdown of different components in Eyeriss and SpinalFlow. The filter buffer and scratchpads are the dominant energy contributors in SpinalFlow and Eyeriss respectively. In SpinalFlow, no access is made to the filter buffer (which contributes 88% of total energy) whenever a zero-valued activation is encountered. Due to this, energy of SpinalFlow scales well with sparsity. Eyeriss on the other hand has to access its GLB and ifmap-spad (together contribute 44% of total energy), irrespective of activation sparsity. Therefore, the gap between SpinalFlow and Eyeriss grows as sparsity increases. Figure 4.11 also shows that SpinalFlow is sub-optimal when handling DWC

layers. This is because ofmaps in DWC do not share inputs, so the 128-wide PEs and buffer fetches are severely under-utilized.

Figure 4.13 shows the energy/inference of SpinalFlow relative to Eyeriss for our three full workloads. MobileNet is a combination of 13 DWC and 13 PWC layers. Even though SpinalFlow is inefficient at processing DWC layers at high resolution, it is overall more energy-efficient than Eyeriss for MobileNet because the DWC layers account for only 3% of execution time. The energy savings are generally higher for the other two workloads. Unlike Spiking-Eyeriss, SpinalFlow is more energy-efficient than Eyeriss at nearly all evaluated sparsity/resolution points. At 4-bit resolution and 90% sparsity, on average for the three full workloads, SpinalFlow consumes 5× less energy than the Eyeriss baseline.

Note that SNNs naturally exhibit high sparsity [109]. Prior work [63] shows that t-SNNs trained with STDP can achieve significantly higher sparsity at lower input resolutions than ANNs. While ANNs are unlikely to exhibit higher levels of sparsity than that already shown in prior work and assumed in our ANN baseline, ANNs can certainly operate at lower resolution with lower accuracy We next evaluate how SpinalFlow energy compares against Eyeriss at lower resolutions.

4.5.1.2 Effect of Low Resolution

Figure 4.14 plots the energy per inference of our synthetic workloads on SpinalFlow – unlike earlier graphs that normalize against an 8-bit Eyeriss baseline, the data here is normalized against an Eyeriss baseline with the same resolution as SpinalFlow. The ANN sparsity is 60% throughout. In general, the SpinalFlow improvement is a little lower at lower resolutions – note how the left to right trend is slightly increasing in Figure 4.14, whereas it was clearly decreasing in Figure 4.11. This is primarily because the flip-flops in the baseline Eyeriss PE scale down better than the SRAM filter buffer in SpinalFlow. This pattern is also oberved with full workloads shown in Figure 4.15. At 4-bit resolution and 90% sparsity, on average, SpinalFlow consumes 1.8× less energy than a 4-bit Eyeriss baseline.

4.5.1.3 Latency Comparison

Latency/inference of SpinalFlow normalized to Eyeriss (8-b resolution and 60% activation sparsity) is shown in Figure 4.16. We model two versions of Eyeriss, one with

1K global buffer wires (similar to SpinalFlow) and another with 72 (similar to original Eyeriss). Note that in SpinalFlow, latency changes only with the degree of sparsity, and not with resolution. While DWC is an exception because of its low utilization, the other workloads in SpinalFlow are competitive with Eyeriss at 0% sparsity because they have comparable compute and utilization. At high sparsity levels, SpinalFlow is orders of magnitude faster than Eyeriss because the execution time is a function of the spike train size; at 90% sparsity, the speedup is 5.4× on average for our three full workloads. When dealing with sparse inputs, our baseline Eyeriss already saves energy by gating the ALU, but it does not save time by jumping to the next computation. Accelerators like SCNN [98] are able to save time when dealing with sparse inputs. SCNN adds index generation logic and a crossbar network to achieve this and offers a 2.7× performance improvement for the typical sparsity observed in ANNs. Thus, even with a better baseline like SCNN, SpinalFlow offers a significant speedup [63, 109].

4.5.2 SpinalFlow versus Spiking-Eyeriss

Figure 4.17 shows the energy per inference of SpinalFlow normalized to that of Spiking-Eyeriss at corresponding resolution and degree of sparsity. As both architectures are executing t-SNNs, the computation overhead will be similar for all design points. Recall that unlike SpinalFlow, Spiking-Eyeriss processes spikes tick-by-tick, and incurs significant off-chip and GLB overhead. 70% of the on-chip energy and 64% of the total energy of t-SNNs at 8b resolution is due to GLB accesses and off-chip accesses respectively. This results in Spiking-Eyeriss consuming an average of 35× more energy than SpinalFlow at 8b resolution. Once input resolution is decreased, the overhead of off-chip and GLB accesses reduce significantly and hence the improvement of SpinalFlow over Spiking-Eyeriss reduces as well. For similar reasons, the relative efficiency of SpinalFlow improves at higher degrees of sparsity. Figure 4.18 shows a similar trend on our three workloads. At 4-bit resolution and 90% sparsity, all three workloads on SpinalFlow consume roughly 5× lower energy than Spiking-Eyeriss. Spiking-Eyeriss processes inputs tick-by-tick, and hence is $2^{resolution}$ times slower than Eyeriss. It is therefore multiple orders of magnitude slower than SpinalFlow.

4.5.3 Fully-Connected Layers

For fully-connected networks with a batch size of 1, the execution is entirely dominated by the bottleneck in fetching weights from DRAM, which accounts for 90% of the total system energy in both SpinalFlow and Eyeriss. If we assume that 200 inputs are batched, then SpinalFlow is an order of magnitude more efficient than baseline Eyeriss. This is because baseline Eyeriss has a relatively small on-chip storage capacity, requiring multiple DRAM accesses for either activations or weights (depending on the chosen dataflow). However, if we were to augment Eyeriss with substantial on-chip buffer capacity (similar to that of SpinalFlow) and a dataflow to maximize weight reuse, the energy bottleneck again shifts to the other microarchitectural components in Eyeriss and SpinalFlow.

We observe similar trends as for convolutional layers. Figure 4.19 shows the energy of SpinalFlow with respect to Eyeriss for executing workloads FC-A and FC-B at a batch size of 200. At 0% sparsity, SpinalFlow consumes 20% more energy than Eyeriss, whereas at a higher sparsity level of 90% and at 4b resolution, SpinalFlow consumes 0.3$\times$ of the energy consumed by Eyeriss.

4.5.4 Scalability Study

Next, we analyze the scalability of SpinalFlow as the number of PEs and, correspondingly, the size of the weight buffer are varied. Figure 4.20 shows the change in energy per inference as the PEs (and the weight buffer) are increased from 32 (144 KB) to 512 (2.25 MB) for ResNet. Increasing the compute and storage resources increases the efficiency, but with diminishing returns beyond 128 PEs. This is because few layers use more than 256 feature maps, and weight buffer energy increases significantly. Similar results were observed for other workloads as well.

Figure 4.21 shows the throughput-per-area for our three workloads and SpinalFlow area as the PE count (and buffer size) are varied. Because of the large weight buffer in SpinalFlow, it does not fare as well as Eyeriss in throughput-per-area in many cases. This effect can be alleviated by using fewer PEs and smaller buffers.

4.5.5 SpinalFlow versus SCNN

As a sensitivity analysis, we also compare SpinalFlow to SCNN [98], an ANN accelerator that exploits sparsity. We model SCNN at 8-bit resolution and with 8 PEs (128 MAC

units) for an iso-ALU comparison. We make favorable assumptions for SCNN – we do not model the computation and storage overheads of meta-data (indices), and the crossbar that connects MACs with the accumulator buffer. We model the accumulator buffer with 2 banks instead of 32 due to limitations with Cacti. For ResNet, SpinalFlow at 60% activation sparsity consumes 1.02× less energy than SCNN, whereas at 90% activation sparsity, it consumes 1.16× less energy than SCNN. While we assume a similar buffer organization as in the original SCNN work, we expect that a sweep of different buffer hierarchies may reveal more energy-efficient SCNN design points.

4.6 Re-Visiting the SNN versus ANN Debate

There remains a healthy debate within the community about the merits of SNNs and ANNs. These issues have been discussed in keynotes at ASPLOS 2014 (Gehlhaar [37]), HPCA 2015 (Modha [87]), ISCA 2015 (Temam [125]), ASPLOS 2016 (Williams [131]), and FCRC 2019 (Smith [116]). In this section, we summarize the current state of this debate, given our findings. In particular, our analysis is among the first to demarcate when an SNN is a better or worse choice than an ANN.

4.6.1 Comparing SNN versus ANN Efficiency

A couple of papers have analyzed SNN vs. ANN efficiency. A MICRO 2015 paper by Du et al. [28] attempted a head-to-head comparison of ANN and SNN hardware. They compare a two-layer ANN (100 neurons in the first layer and 10 neurons in the second layer) against a one-layer SNN (300 neurons) on the MNIST workload for digit recognition. The architecture models assume some dedicated hardware per neuron, an approach that does not scale up to large networks. For this limited comparison, the authors conclude that ANNs and SNNs have similar per-neuron area and power overheads. A more recent paper by Khacef et al. [61] improves upon this prior work with more efficient and more accurate neuron models, but draws similar conclusions for a limited set of networks and dataflows. Our analysis of larger/diverse networks and architectures shows that data reuse is a key factor; with baseline dataflows, we show (contrary to prior work) that SNNs are an order of magnitude worse than ANNs in most key metrics. The execution time is $2^{resolution}$ higher and energy is 35× higher at high resolution and 2× higher at low resolution. With our new

dataflow, for smaller input intervals, where t-SNNs are expected to perform best [116], SNNs consume 5× lower energy. SNN and ANN energy efficiency are almost on par even for large input intervals. When networks exhibit high sparsity, also expected for t-SNNs [109, 116], SNN execution time and energy improve significantly. The comparison is more nuanced if ANNs are also allowed to lower resolution; this is an approach that is known to significantly lower accuracy [21, 57, 104, 142]. With this approach, as shown in Figures 4.14 and 4.15, ANNs and SNNs are comparable in terms of energy; the nature of the network and the degree of sparsity determines the winning architecture.

4.6.2 Discussion of Prediction Accuracy

The improved dataflow in this chapter only improves efficiency (time and energy), and we quantify the relationship between efficiency and sparsity/resolution. The new dataflow has no impact on accuracy. But since accuracy is a primary consideration in the SNN vs. ANN debate, for completeness, we articulate the conditions under which SNNs or ANNs may be superior.

First, consider a use-case where labeled datasets are available for supervised training on GPU/TPU clusters. This is the scenario where ANNs with back-propagation based SGD represent the state-of-the-art. A number of studies have shown that r-SNNs can borrow such weights and achieve similar accuracies as ANNs [28, 31, 32, 61, 107]. This is primarily because an r-SNN neuron can emulate the behavior of an ANN. On the other hand, t-SNN training has received less attention and t-SNNs generally have lower accuracies. We summarize some of these key results in Table 4.4. Given that t-SNNs are more efficient in terms of time and energy on SpinalFlow, t-SNN training is an area that demands future investment, a point also made by Smith [116]. The work of Comsa et al. [20] shows an example t-SNN operating at low resolution and high sparsity that matches the accuracy of an ANN. With further advances along these lines, t-SNNs may be able to achieve higher accuracies and lower energy than high- and low-res ANNs. Note that low resolution has typically been a significant handicap for ANNs in terms of accuracy, but this is not the case for SNNs [63]. For a more complete summary of the trade-off space, we also show the impact of low-resolution ANNs on accuracy in Table 4.4, e.g., note that a 4-bit ANN can reduce accuracy by 2.9% (for AlexNet on ImageNet [142]).

A second use case is one where continual learning is required. The ability of STDP to efficiently perform online training allows SNNs to react faster when new inputs are encountered, e.g., new landscapes during disaster recovery or new accents during speech processing. Note that in such use cases, curated, pre-processed, and labeled datasets are often not available. The training may also have to be performed at low energy on an edge device, e.g., a rover handling disaster recovery. The area of continual learning [17] is an emerging one with a limited amount of literature. ANNs trained with SGD suffer from the concept of catastrophic forgetting [82, 105] when they are sequentially trained on two datasets, i.e., SGD's global error minimization tends to perturb all network parameters to react to the new dataset [6, 65, 75]. On the other hand, STDP does not require labeled datasets and its localized training can naturally earmark a subset of neurons for the new dataset, while not perturbing the rest of the model [6]. For such use cases, SNN/STDP is a clear winner and can exploit the new dataflows to significantly reduce execution time and energy.

Our results also serve as a useful guideline for researchers developing SNNs for various use cases. Our analysis quantifies the scenarios (resolution, sparsity, network topology) under which SNNs can surpass the energy efficiency of ANNs. We highlight the need to develop accurate training models for t-SNNs because it results in higher sparsity levels and lower energy per inference.

Table 4.1. Parameters for our ANN (based largely on Eyeriss) and baseline SNN.

Components	Eyeriss	Spiking-Eyeriss
PE Array	12 ×14	12 ×14
ALU per PE	8-b FxP MAC	8-b FxP Add & Cmp
Filter scratchpad	224× 8-b	224× 8-b
psum scratchpad	24×8-b	24×8-b
ifmap scratchpad	12×8-b	12×1-b
Global Memory	54 KB	54 KB
Core frequency	200 MHz	200 MHz
Off-chip memory	HBM2	HBM2

Table 4.2. Workloads, degree of sparsity, and resolution. SC - Standard Conv, DWC - Depth-Wise separable Conv, PWC - Point-Wise separable Conv. The SNN network from [63] will be referred to as STDP-Net for the rest of the chapter.

ResNet MobileNet	33 convolutional layers, 1 FC layer 13 PWC and 13 DWC layers
STDP-Net	2 conv layered network from [63]
SC-A SC-B	3x3x64x64, 1 layer, Synthetic 3x3x512x512, 1 layer, Synthetic
DWC-A DWC-B	3x3x1x64, 1 layer, Synthetic 3x3x1x512, 1 layer, Synthetic
PWC-A PWC-B	1x1x64x64, 1 layer, Synthetic 1x1x512x512, 1 layer, Synthetic
FC-A FC-B	4096x4096, 1 layer, Synthetic 1024x1024, 1 layer, Synthetic
Sparsity	60%, 90%, 98%
Resolution	8b, 6b, 4b, 3b, 2b

Table 4.3. Architecture specifications of SpinalFlow and Eyeriss-1K. FB- Filter Buffer, GLB - Global Buffer, B/W - Bandwidth, A - Area in mm^2, P - Power in mW

Components	Eyeriss-1K 8b(4b)	SpinalFlow 8b(4b)
PEs	168	128
ALU/PE	8b (4b) MAC	8b(4b) Add, Cmp
Filt scratchpad	224× 8b (4b)	1× 8b (4b)
psum/V_{mem} spad	24×8b (4b)	1×8b (4b)
ifmap scratchpad	12×8b (4b)	1×8b (4b) (shared)
Global Buffer	54 (27) KB	585 (292.5) KB
GLB bus-width	448(224)-psum, 448(224)-filt,112(56)-ifmap	1024(512)-filt, 8(4)-spike
Core frequency	200 MHz	200 MHz
DRAM B/W	30 GB/sec	30 GB/sec
PEs A/P	0.353(0.1412)/515.5	0.024(0.012)/51.5
Min find A/P	-	0.002(0.00092)/1
Inp Buff A/P	-	0.069(0.0088)/4.3
GLB/FB A/P	0.715(0.21)/48.7	1.99(0.78)/105.6
Total A/P	1.068(0.35)/564.2	2.09(0.801)/162.4

Table 4.4. Accuracy comparison with supervised training on labeled datasets.

Workload	ANN Accuracy	SNN Accuracy
MNIST	99.8% [129] 1-bit res: 99.04% [21]	r-SNN(SGD): 99.59% [73] r-SNN(STDP+SGD): 99.28% [72] t-SNN (SGD): 97.96% [20] t-SNN (STDP): 98.4% [63]
CIFAR10	92.38% [107] 1-bit res: 88.6% [21]	r-SNN: 90.53% [133]
AlexNet on ImageNet	55.9% [51, 142] 1-bit res: 44.2% [104] 2-bit res: 49.8% [142] 4-bit res: 53.0% [142]	r-SNN: 51.8% [51]
VGG on ImageNet	70.52% [109]	r-SNN: 69.96% [109]
ResNet on ImageNet	70.69% [109]	r-SNN: 65.47% [109]

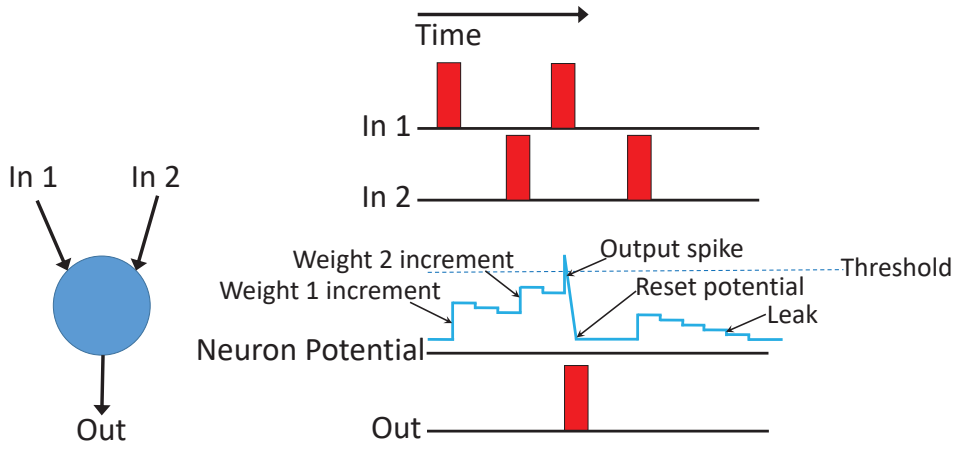

Figure 4.1. A basic 2-input LLIF spiking neuron. The figure shows how the neuron potential is incremented when input spikes are received, how a leak is subtracted when there are no input spikes, and how an output spike is produced when the potential crosses the threshold.

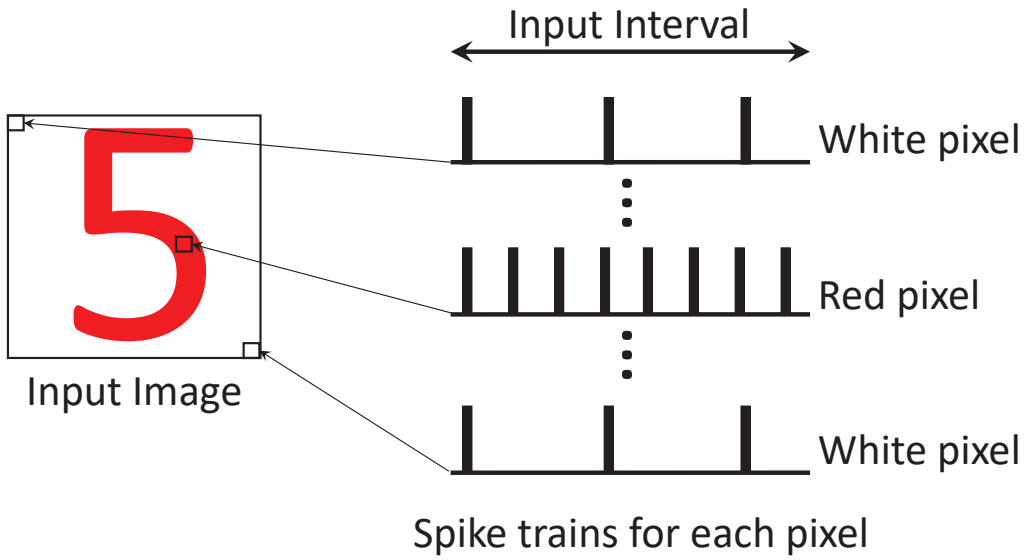

Figure 4.2. Example of an input image converted into a number of input spike trains that are fed to a rate-coded SNN (r-SNN).

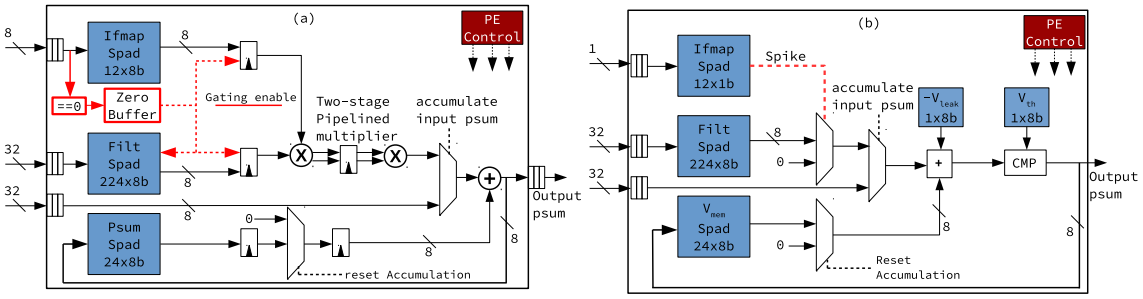

Figure 4.3. Architectures Evaluated. (a) PE in Eyeriss [14]. (b) PE in Spiking-Eyeriss.

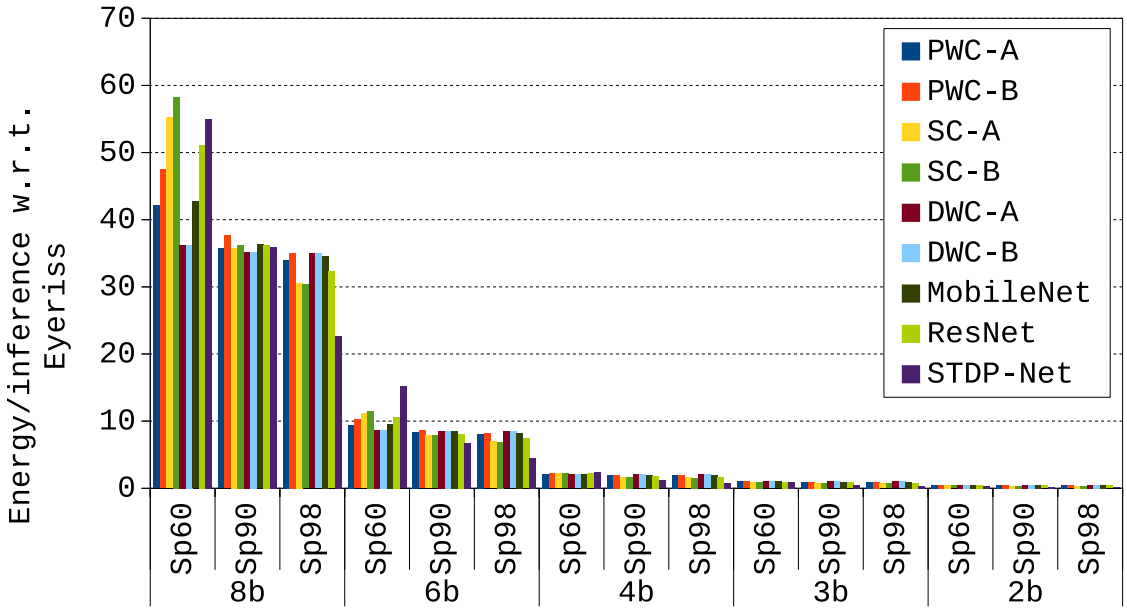

Figure 4.4. Energy consumed per inference by r-SNN on Spiking-Eyeriss normalized to Eyeriss. Sp60, Sp90, and Sp98 refers to 60%, 90%, and 98% sparsity respectively.

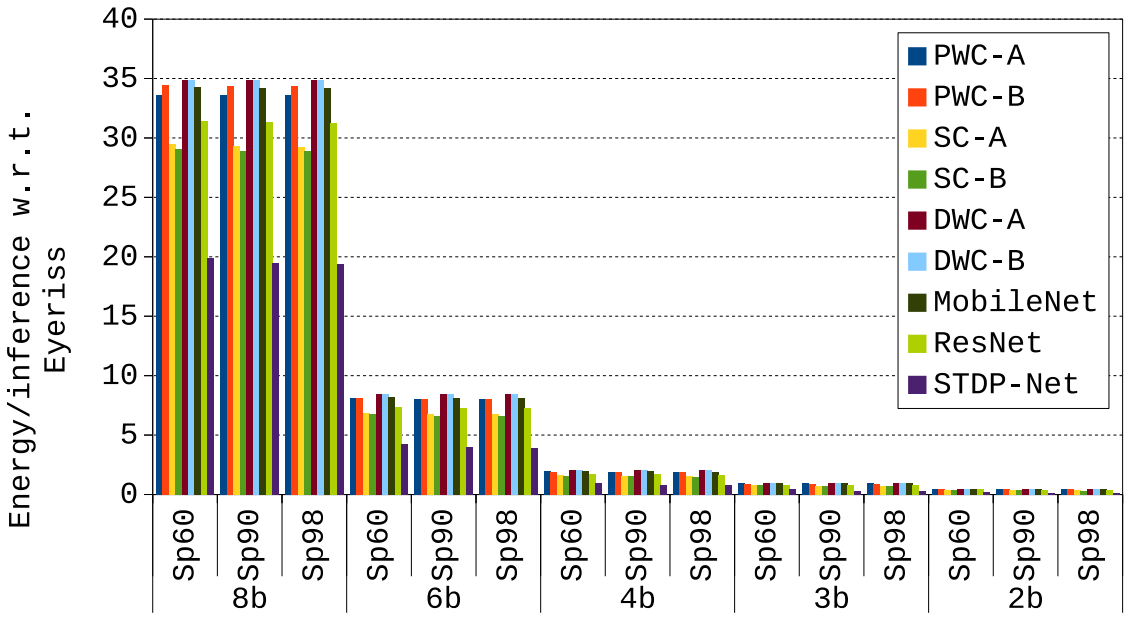

Figure 4.5. Energy consumed per inference by t-SNN on Spiking-Eyeriss normalized to Eyeriss.

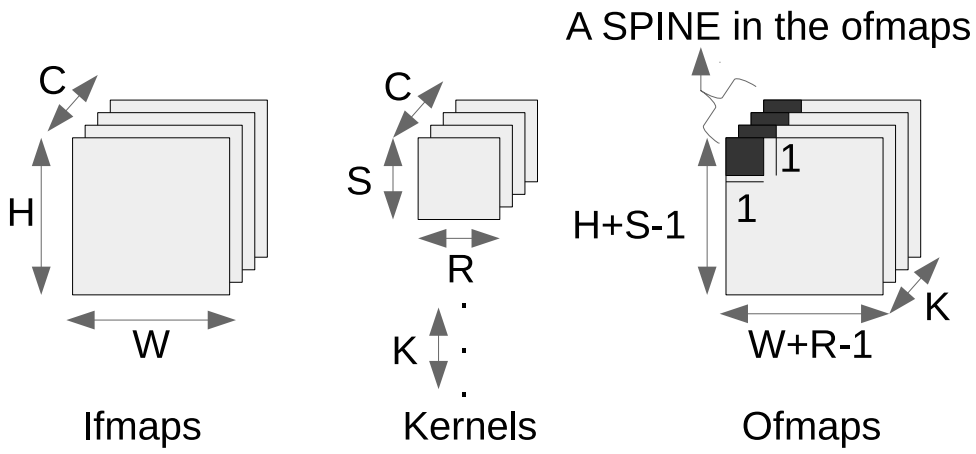

Figure 4.6. A spine in a CONV layer.

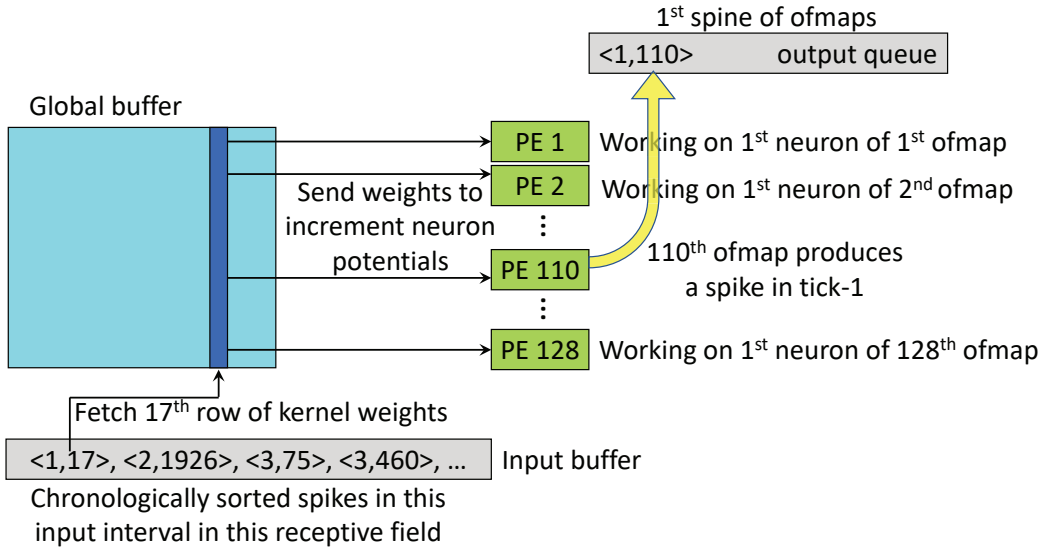

Figure 4.7. Example: Step1, Cycle 1.

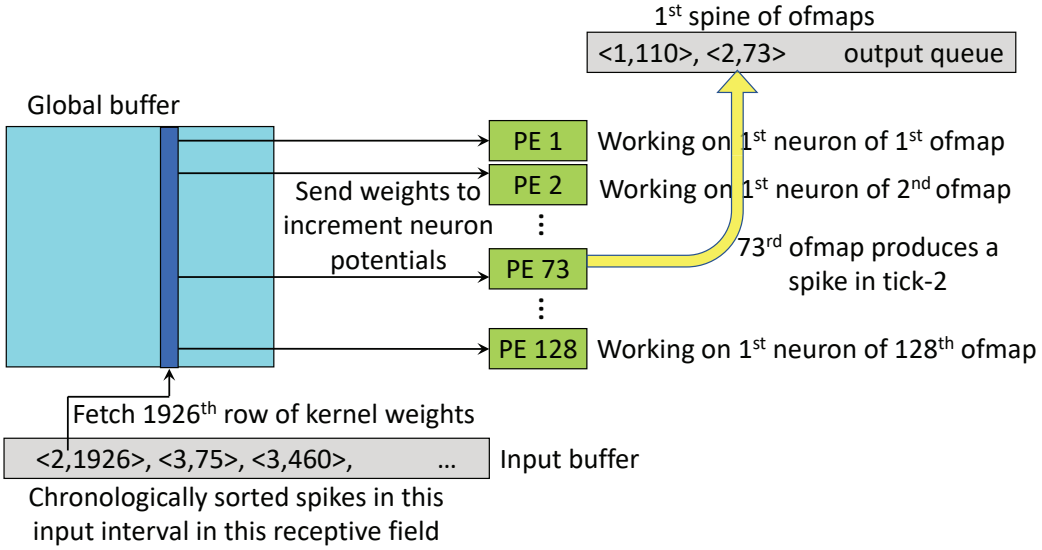

Figure 4.8. Example: Step1, Cycle 2. Step 1 continues until all receptive field entries (up to 2K) have been processed.

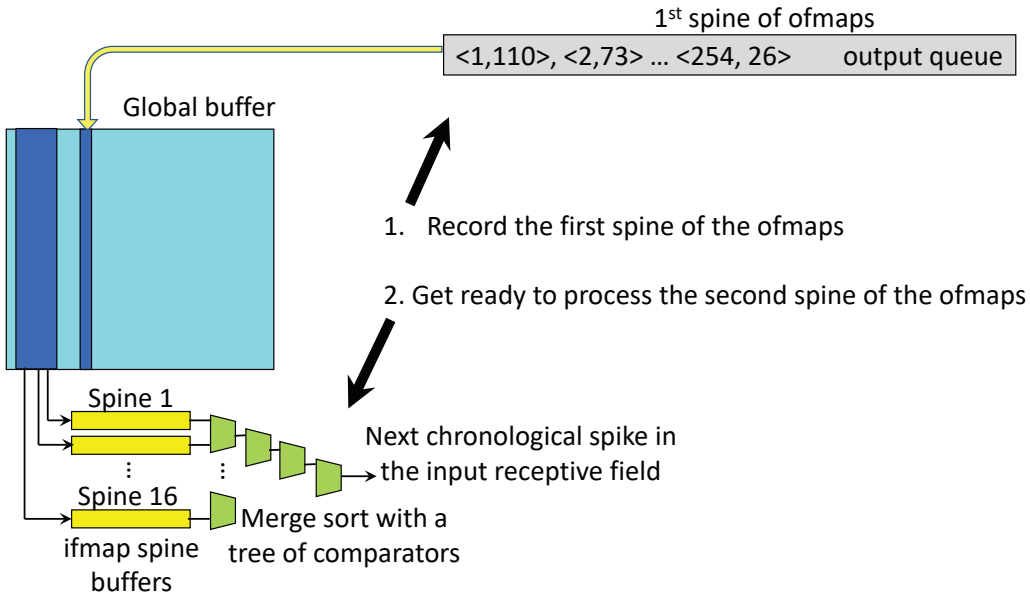

Figure 4.9. Example: End of Step 1 and set-up before Step 2.

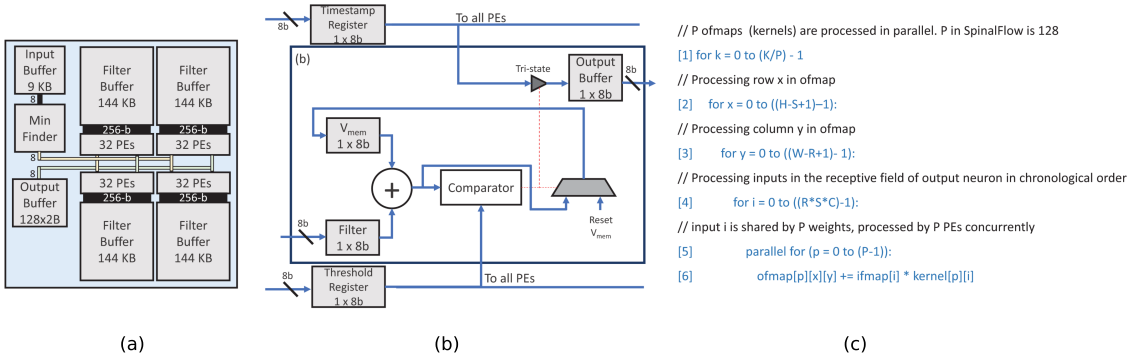

Figure 4.10. SpinalFlow. (a) Chip. (b) PE details. (c) Dataflow pseudocode.

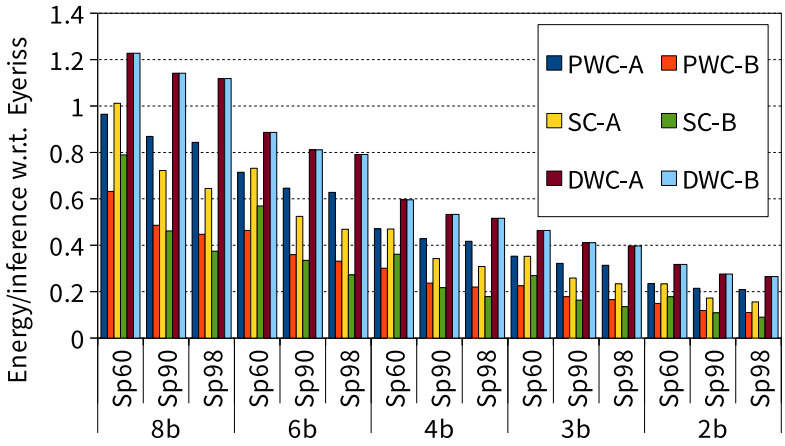

Figure 4.11. Energy/inference of SpinalFlow normalized to an 8-bit Eyeriss with 60% sparsity. Sp60, Sp90, and Sp98 refers to 60%, 90% and 98% sparsity for the SNN.

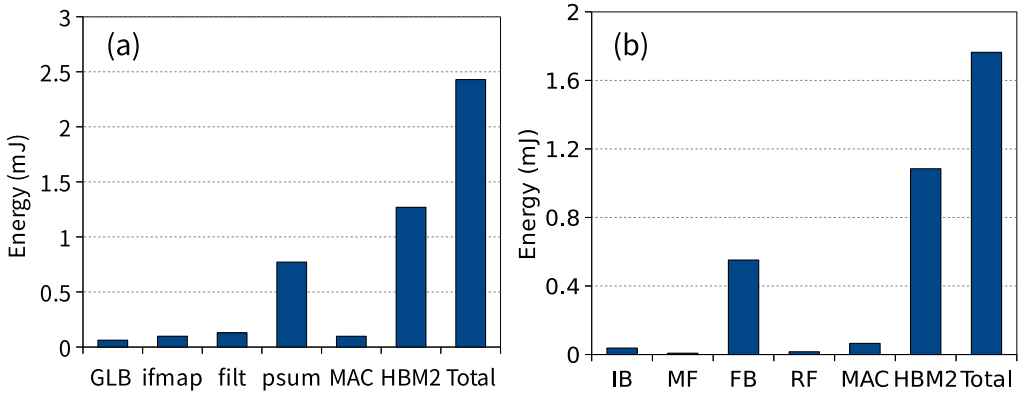

Figure 4.12. Energy/inference of ResNet at 8b resolution and 60% sparsity. (a) On Eyeriss, (b) On SpinalFlow. ifmap, filt and psum refers to corresponding scratch-pads in Eyeriss PE.

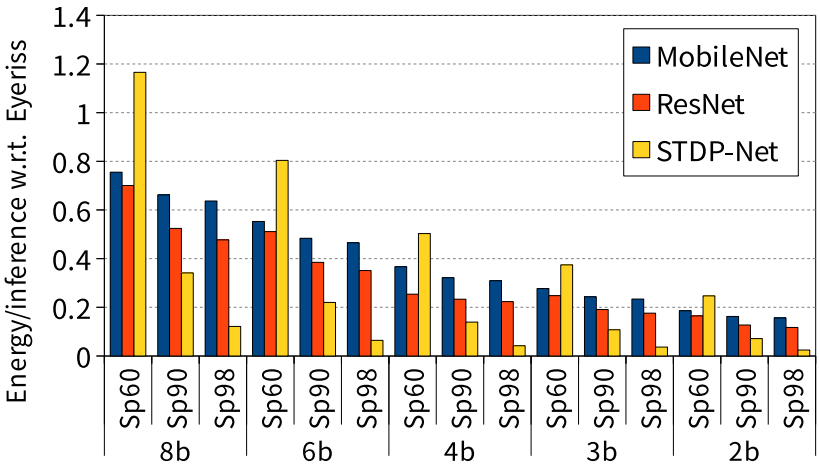

Figure 4.13. Energy per inference for SpinalFlow for full workloads, normalized to 8bSp60 Eyeriss.

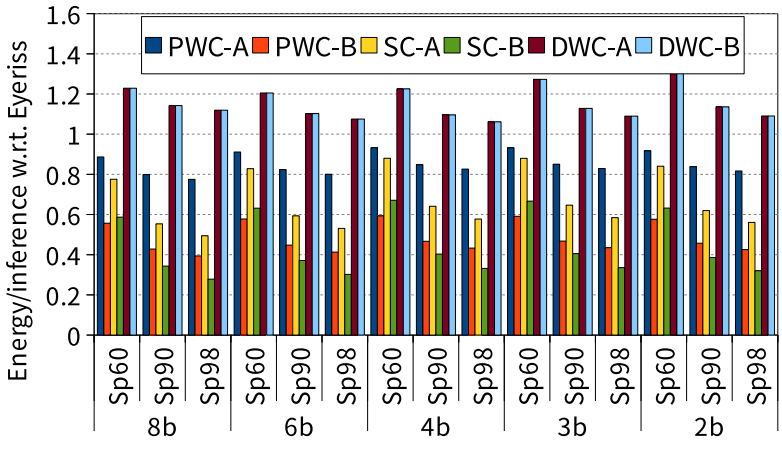

Figure 4.14. Energy/inference of SpinalFlow, normalized to an Eyeriss baseline with the same resolution as SpinalFlow. Note that sparsity of Eyeriss is fixed at 60%.

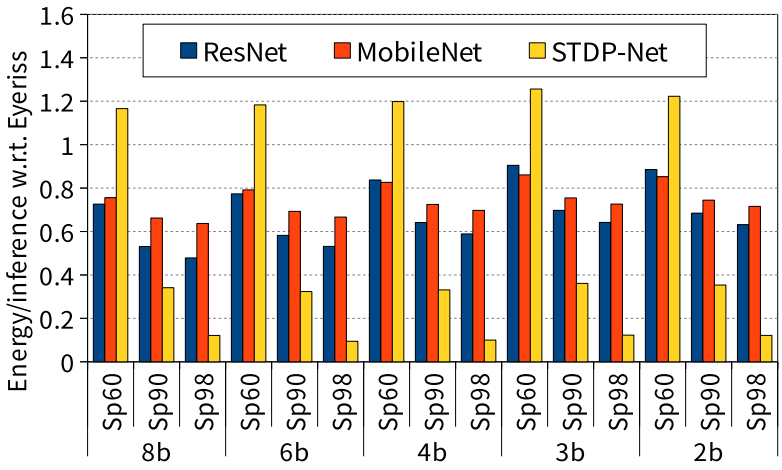

Figure 4.15. Energy/inference of SpinalFlow for full workloads, normalized to an Eyeriss baseline with the same resolution as SpinalFlow.

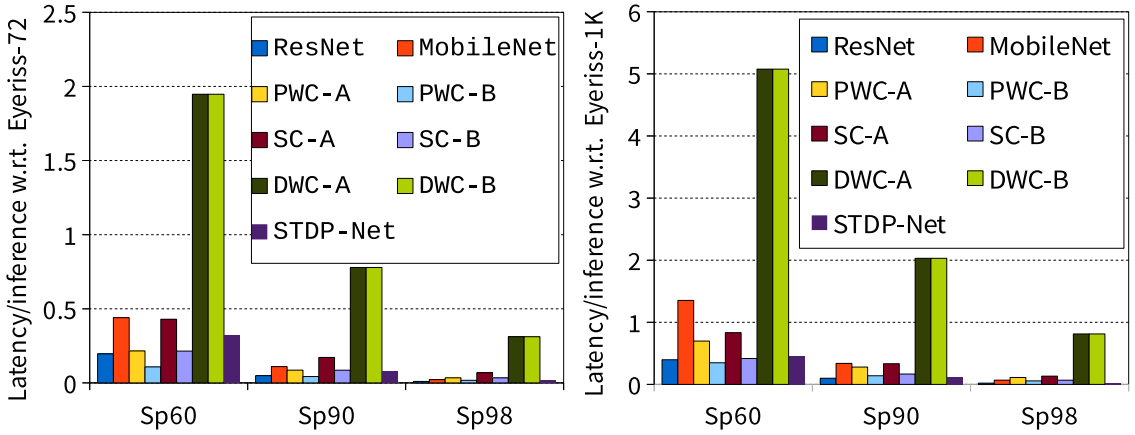

Figure 4.16. Latency per inference of SpinalFlow with respect to Eyeriss (a) with 72 GLB links and (b) with 1024 GLB links.

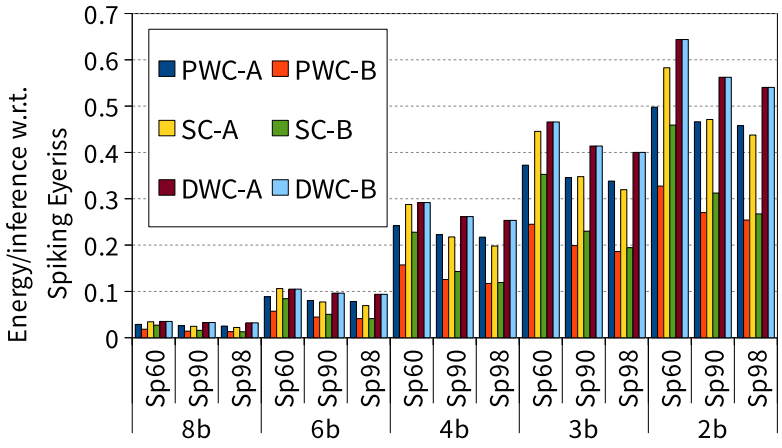

Figure 4.17. SpinalFlow energy per inference normalized to Spiking-Eyeriss for synthetic conv workloads.

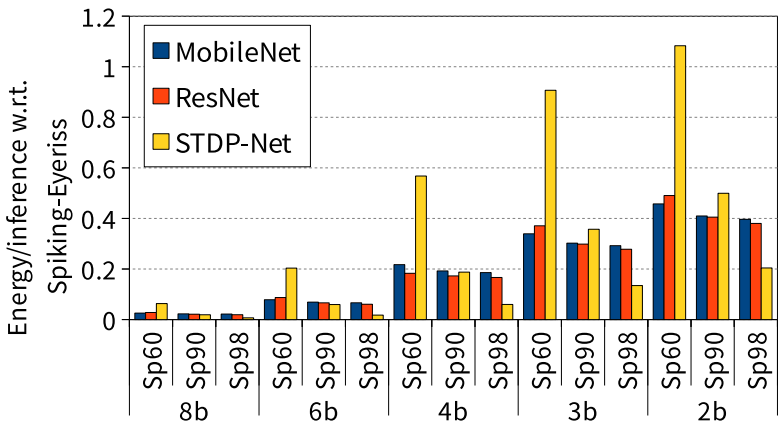

Figure 4.18. SpinalFlow energy per inference normalized to Spiking-Eyeriss for full network workloads

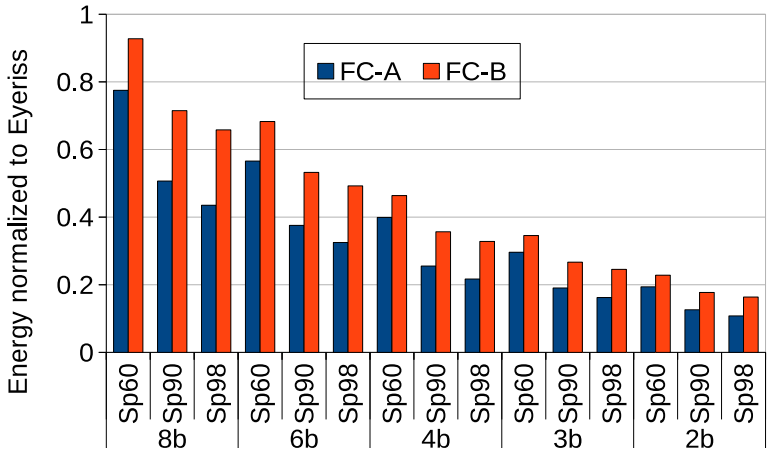

Figure 4.19. Energy per Inference of the synthetic fully connected workloads for SpinalFlow normalized to Eyeriss.

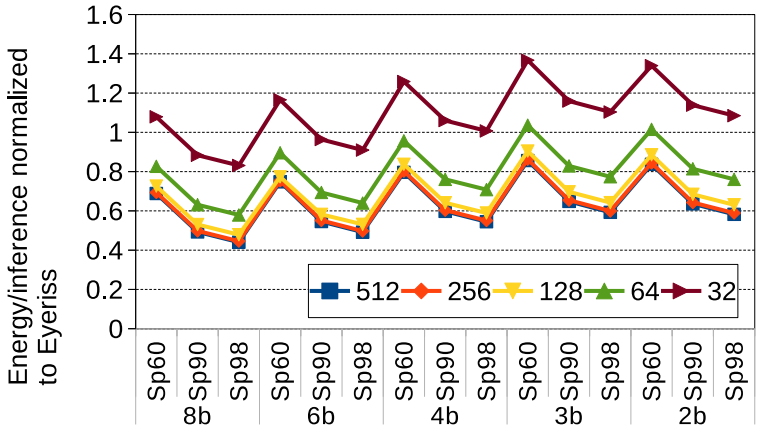

Figure 4.20. Energy per inference for ResNet on SpinalFlow, normalized to Eyeriss, as a function of the number of PEs in SpinalFlow.

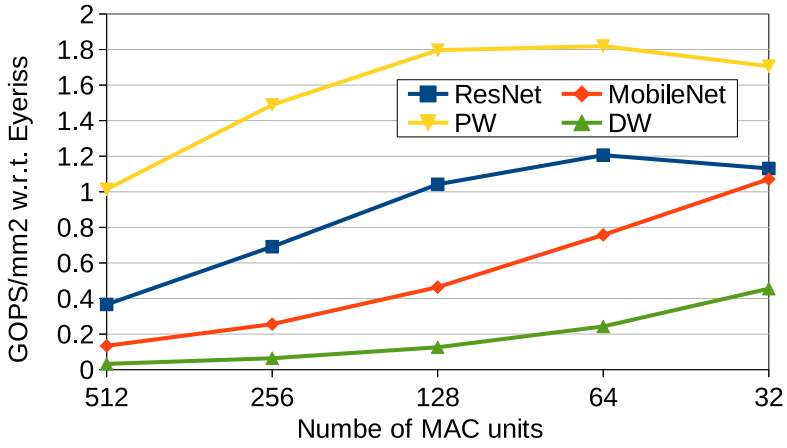

Figure 4.21. Compute density ($GOPS/mm^2$) of SpinalFlow normalized to Eyeriss and its area as the number of PEs is varied.

CHAPTER 5

CONCLUSIONS AND FUTURE WORK

The need for efficient deep neural network training solutions will increase as the applications of machine learning grow. As the tasks become more complex, the model size increases, which increases the overhead of training. Furthermore, labeled datasets may not be available for all complex tasks. In this dissertation, we explore two different avenues to tackle these problems. First, we explore well established concepts like partitioning and pipelining strategies in distributed training. Second, we investigate and quantify the potential and possibility of using spiking neural networks to reduce training energy. In the next three sections, I conclude this dissertation with a high-level overview and key insights from each of the three projects.

5.1 Criticality-Aware Pipeline-Parallel Training

In this work, we conduct an in-depth analysis of the energy and performance of various execution strategies for DNN training. Specifically, we explore the design space of pipelining and partitioning strategies, and their interaction with batch size, group count, chip count, and workload. Contrary to existing knowledge, we observe that convolutional networks achieve optimal performance without grouping at large batch size, and fully connected networks achieve optimal performance with grouping and data parallelism at large batch size.

We also investigate the impact of spatial and temporal pipelining within a chip once the cluster-level execution strategy is defined. Naively adopting a chip-level spatial pipeline leads to slowdown and inefficiency. To overcome this we propose Cafine, a Criticality-Aware Fine-grained Pipeline. On average, Cafine achieves throughput similar to PipeDream, while consuming 4% less energy for CNN workloads and 8% less energy for FC workloads. We conclude that there is a rich space of optimal execution strategies that must be considered for each workload and deployment.

5.2 Re-RAM Based Analog In-situ SNN Accelerator

In INXS, we show that the use of a memristor crossbar can significantly accelerate the computations required by SNNs. The resulting architecture not only removes the constraints posed by the state-of-the-art TrueNorth architecture, it also surpasses TrueNorth on all metrics by one to three orders of magnitude. The INXS architecture takes a significant step in bridging the current large gap between ANN and SNN accelerators.

5.3 Digital SNN Accelerator and Dataflow

Our final work first shows that the baseline SNN architecture, Spiking Eyeriss, is severely penalized by repeated accesses to neuron potential and filter weights as ticks are sequentially processed in the input interval. The Spiking-Eyeriss design consumes $2\times$ more energy than baseline Eyeriss, even at high sparsity and low resolution. It is also $2^{resolution}$ times slower than Eyeriss. We then devised a new architecture and dataflow that increases data reuse and is tailored for the high sparsity that is expected in future SNNs.

The resulting SpinalFlow design improves energy efficiency by $5\times$ over Eyeriss and by $1.8\times$ over a 4-bit version of Eyeriss. It consumes less energy than Eyeriss at most evaluated sparsity/resolution points. The new designs are effective for a range of convolutional layers, and even more effective for memory-constrained fully-connected layers. In terms of performance, SpinalFlow is faster than Eyeriss by $5.4\times$, when assuming a sparsity level of 90%. Because SpinalFlow's weight accesses are less regular, it needs a larger buffer for weights, and yields lower $throughput/mm^2$ than Eyeriss for some workloads. The new architecture also greatly improves the energy,latency, and throughput for accelerators, like TrueNorth, that will be used to simulate brain models [4, 74].We thus show that for large neural networks, reuse management and sparsity exploitation are key in determining SNN vs. ANNrelative efficiency.

Our results also serve as a useful guideline for researchers developing SNNs for various use cases. Our analysis quantifies the scenarios (resolution, sparsity, network topology)under which SNNs can surpass the energy efficiency of ANNs.We highlight the need to develop accurate training models for t-SNNs because it results in higher sparsity levels and lower energy per inference.

5.4 Support for STDP-Based Training in INXS and SpinalFlow

Both the SNN accelerators proposed, INXS and SpinalFlow, are inference accelerators. They do not innately support STDP-based training. With STDP being a localized online training technique, support STDP in SpinalFlow (and INXS) can be achieved with the introduction of minimal hardware units.

5.4.1 STDP-Based Training

The two key computations involved in STDP-based training are calculation of Δt and updating the weights based on Δt. Δt can be calculated by computing the difference between the timing of a neuron's spike and that of it's input spike. Let w_{ij} be the synaptic weight between neuron i in layer l and neuron j in layer $l-1$. Δt between neuron i and j is given by the following equation:

$$\Delta t = t_j - t_i$$

where t_j is spike time of neuron j, and t_i is the spike time of neuron i.

Updating weight w_{ij} after Δt is calculated is given by the equation:

$$w_{ij}^{t+1} = \begin{cases} w_{ij}^{t} + \mu e^{\Delta t}, & \text{if } \Delta t \geq 0 \\ w_{ij}^{t} - \mu e^{\Delta t}, & \text{if } \Delta t \leq 0 \end{cases}$$

5.4.2 Hardware Support for STDP in SpinalFlow

Next, we discuss the additions required to SpinalFlow in order to support STDP-based training. As mentioned earlier, the key computations required to perform STDP are calculation of Δt and performing weight update. In order to calculate Δt, the time stamps of input spikes to a neuron needs to compared with the time stamp of its spike. The output spike generated is already stored in a buffer in a sorted fashion (as discussed in section 4.4). As the comparison can only be performed after output spikes are generated, the input spikes also need to be stored till then. We introduce a new structure to SpinalFlow, the input spike buffer, to handle this. The input spike buffer stores the sorted input spikes which are the output of min finder. Once an output spine is generated, the comparison can be performed. In order to do this, we introduce functional units within PEs such that forward pass can proceed in parallel. Thus, we estimate that, the minimal additions

required to SpinalFlow in order to handle STDP are 1. input spike buffer and 2. subtractor. Table 5.1 lists the are of different hardware components in SpinalFlow without and with STDP support. The two new components increase area of SpinalFlow by 4%. Note that this is the minimal hardware changes required to support STDP. We leave designing STDP optimized accelerator as future work.

5.5 Concluding Remarks

The key to unlocking the full potential of SNNs involves addressing the challenges of both designing efficient architectures and effective training methods. In this dissertation we focused on the first challenge, and introduced novel dataflow/architecture that push the state-of-the-art for SNN acceleration and also make SNNs competitive to ANNs. With SpinalFlow we showed that efficient spike representation, sparsity handling, and reuse management are necessary to achieve high performance at low energy for SNNs. Compared to ANNs, SNNs achieve better energy efficiency at high sparsity and low resolution (potential of SNNs). Motivated by several in-situ analog ANN accelerators, we design INXS, an analog in-situ accelerator for SNNs. The highly parallel and low cost computations enabled by memristor crossbars result in an architecture that provides orders of magnitude better performance and efficiency than commercial products like IBM TrueNorth. In spite of this promise, the lack of commercial success of in-situ analog arithmetic products, and the need for complex structures to efficiently handle sparsity makes INXS a less attractive alternative to SpinalFlow for SNN acceleration. Tangentially, we also investigated distributed training strategies that could reduce training energy. Our design space exploration uncovered scenarios in distributed training where the optimal parallelizing strategy diverges from existing assumptions. We also introduced Cafine, which improves training efficiency by intelligently deciding between spatial and temporal pipelining.

5.6 Future Work

The next crucial step towards realizing unsupervised efficient training of SNNs is to devise powerful and robust training techniques. The training technique needs to be online, unsupervised and also localized (like STDP) in order to facilitate both real-time and

efficient learning. Both SpinalFlow and INXS need to be equipped with structures to facilitate STDP based training. This requires support for buffering neuron spike-times and for performing parameter updates. As opposed to back-propagation based training, parameter updates in STDP are localized to a layer. This might also necessitate new parallelization strategies as existing ones are optimized for SGD based training methods. With ASICs specialized for DNNs becoming common in modern mobile devices, and the potential of efficient (and online) training enabled by SNNs, on-device learning can become a reality (especially while the device is plugged-in). In such a setting, heterogeneous (CPUs, GPUs, ASICs, etc.) distributed training techniques and dynamic resource management are imperative to fully and efficiently utilize the available resources.

Table 5.1. Area (in mm^2) of SpinalFlow without and with support for STDP-based training.

Components	SpinalFlow	SpinalFlow with STDP support
PE	0.024	0.031
Min find	0.002	0.002
Inp Buffer	0.069	0.069
Inp Spike Buffer	-	0.069
Filter buffer	1.99	1.99
Total area	2.09	2.16

REFERENCES

[1] "Amazon EC2 Trn1 instances," 2021, https://aws.amazon.com/ec2/instance-types/trn1/.

[2] "Dojo, Tesla AI Day." 2021, https://youtu.be/j0z4FweCy4M?t=6340.

[3] K. Ahmed, A. Shrestha, Q. Qiu, and Q. Wu, "Probabilistic Inference Using Stochastic Spiking Neural Networks on a Neurosynaptic Processor," in *Proceedings of IJCNN*, 2015.

[4] F. Akopyan, J. Sawada, A. Cassidy, R. Alvarez-Icaza, J. Arthur, P. Merolla, N. Imam, Y. Nakamura, P. Datta, G. Nam, B. Taba, M. Beakes, B. Brezzo, J. Kuang, R. Manohar, W. Risk, B. Jackson, and D. Modha, "TrueNorth: Design and Tool Flow of a 65mW 1 Million Neuron Programmable Neurosynaptic Chip," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 34(10), 2015.

[5] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. Jerger, and A. Moshovos, "Cnvlutin: Zero-Neuron-Free Deep Convolutional Neural Network Computing," in *Proceedings of ISCA-43*, 2016.

[6] J. M. Allred and K. Roy, "Stimulating STDP to Exploit Locality for Lifelong Learning without Catastrophic Forgetting," in *arXiv preprint arXiv:1902.03187*, 2019.

[7] M. Alwani, H. Chen, M. Ferdman, and P. Milder, "Fused-Layer CNN Accelerators," in *Proceedings of MICRO*, 2016.

[8] B. Benjamin, P. Gao, E. McQuinn, S. Choudhary, A. Chandrasekaran, J. Bussat, R. Alvarez-Icaza, J. Arthur, P. Merolla, and K. Boahen, "Neurogrid: A Mixed-Analog-Digital Multichip System for Large-Scale Neural Simulations," *Proceedings of the IEEE*, vol. 102(5), 2014.

[9] Y. Cao, Y. Chen, and D. Khosla, "Spiking Deep Convolutional Neural Networks for Energy-Efficient Object Recognition," *International Journal of Computer Vision*, vol. 113(1), 2015.

[10] Cerebras, "Cerebras Wafer Scale Engine: An Introduction," 2019, https://www.cerebras.net/wp-content/uploads/2019/08/Cerebras-Wafer-Scale-Engine-Whitepaper.pdf.

[11] ——, "Training Giant Neural Networks Using Weight Streaming on Cerebras Wafer-Scale Systems," 2021, https://f.hubspotusercontent30.net/hubfs/8968533/Virtual%20Booth%20Docs/CS%20Weight%20Streaming%20White%20Paper%20111521.pdf.

[12] ——, "Deep Learning at Scale with the Cerebras CS-1," 2029, https://hotchips.org/assets/program/tutorials/HC2020.Cerebras.NataliaVassilieva.v02.pdf.

[13] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "DianNao: A Small-Footprint High-Throughput Accelerator for Ubiquitous Machine-Learning," in *Proceedings of ASPLOS*, 2014.

[14] Y.-H. Chen, T. Krishna, J. Emer, and V. Sze, "Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks," *IEEE Journal of Solid-State Circuits*, no. 52(1), 2016.

[15] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks," in *Proceedings of ISCA-43*, 2016.

[16] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam, "DaDianNao: A Machine-Learning Supercomputer," in *Proceedings of MICRO-47*, 2014.

[17] Z. Chen and B. Liu, *Lifelong Machine Learning*. Morgan & Claypool, 2018.

[18] P. Chi, S. Li, Z. Qi, P. Gu, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie, "PRIME: A Novel Processing-In-Memory Architecture for Neural Network Computation in ReRAM-based Main Memory," in *Proceedings of ISCA-43*, 2016.

[19] D. Ciresan, U. Meier, and J. Schmidhuber, "Multi-Column Deep Neural Networks for Image Classification," in *Proceedings of CVPR*, 2012.

[20] I. Comsa, K. Potempa, L. Versari, T. Fischbacher, A. Gesmundo, and J. Alakuijala, "Temporal Coding in Spiking Neural Networks with Alpha Synaptic Function," *arXiv preprint arXiv:1907.13223v2*, 2019.

[21] M. Courbariaux and Y. Bengio, "BinaryNet: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1," 2016, arXiv preprint 1602.02830.

[22] C.Yakopcic, M.Z.Alom, and T.M.Taha, "Memristor Crossbar Deep Network Implementation Based on a Convolutional Neural Network," in *Proceedings of IJCNN*, 2016.

[23] Z. Dai, Z. Yang, Y. Yang, J. Carbonell, Q. V. Le, and R. Salakhutdinov, "Transformer-xl: Attentive Language Models Beyond a Fixed-Length Context," *arXiv preprint arXiv:1901.02860*, 2019.

[24] M. Davies, N. Srinivasa, T.-H. Lin, G. Chinya, Y. Cao, S. H. Choday, G. Dimou, P. Joshi, N. Imam, S. Jain *et al.*, "Loihi: A Neuromorphic Manycore Processor with On-Chip Learning," *IEEE Micro-38*, 2018.

[25] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, Q. Le, M. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Ng, "Large Scale Distributed Deep Networks," in *Proceedings of Neural Information Processing Systems*, 2012.

[26] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of Deep Bidirectional Transformers for Language Understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[27] P. Diehl, D. Neil, J. Binas, M. Cook, S. Liu, and M. Pfeiffer, "Fast-Classifying, High-Accuracy Spiking Deep Networks Through Weight and Threshold Balancing," in *Proceedings of IJCNN*, 2015.

[28] Z. Du, D. Rubin, Y. Chen, L. He, T. Chen, L. Zhang, C. Wu, and O. Temam, "Neuromorphic Accelerators: A Comparison Between Neuroscience and Machine-Learning Approaches," in *Proceedings of MICRO-48*, 2015.

[29] L. Durant, O. Giroux, M. Harris, and N. Stam, "Inside Volta: The World's Most Advanced Data Center GPU," 2017, https://devblogs.nvidia.com/inside-volta/.

[30] J. E.Smith, "Space-Time Algebra: A Model for Neocortical Computation," in *Proceedings of ISCA*, 2018.

[31] S. Esser, R. Appuswamy, P. Merolla, J. Arthur, and D. Modha, "Backpropagation for Energy-Efficient Neuromorphic Computing," in *Proceedings of NIPS*, 2015.

[32] S. Esser, P. Meroll, J.V.Arthur, A.S.Cassidy, R.Appuswamy, A. Andreopoulos, D. Berg, J. McKinstry, T.Melano, D. Barch, C. Nolfo, P. Datta, A. Amir, B. Taba, M. Flickner, and D. Modha, "Convolutional Networks for Fast, Energy-Efficient Neuromorphic Computing," in *arXiv*, 2016.

[33] S. Fan, Y. Rong, C. Meng, Z. Cao, S. Wang, Z. Zheng, C. Wu, G. Long, J. Yang, L. Xia *et al.*, "DAPPLE: A Pipelined Data Parallel Approach for Training Large Models," in *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2021.

[34] D. Feldman, "The Spike Timing Dependence of Plasticity," *Neuron*, no. 75(4), 2012.

[35] J. Fowers, K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi *et al.*, "A Configurable Cloud-Scale DNN Processor for Real-Time AI," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018, pp. 1–14.

[36] K. Fukushima and S. Miyake, "Neocognitron: A self-organizing neural network model for a mechanism of visual pattern recognition," pp. 267–285, 1982.

[37] J. Gehlhaar, "Neuromorphic Processing: A New Frontier in Scaling Computer Architecture," 2014, keynote at ASPLOS.

[38] S. Ghodrati, B. Ahn, J. Kim, S. Kinzer, B. Yatham, N. Alla, H. Sharma, M. Alian, E. Ebrahimi, N. Kim, C. Young, and H. Esmaeilzadeh, "Planaria: Dynamic Architecture Fission for Spatial Multi-Tenant Acceleration of Deep Neural Networks," in *Proceedings of MICRO*, 2020.

[39] P. Goyal, P. Dollár, R. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He, "Accurate, Large Minibatch SGD: Training Imagenet in 1 Hour," *arXiv preprint arXiv:1706.02677*, 2017.

[40] B. Graham, "Fractional Max-Pooling," *arXiv preprint arXiv:1412.6071*, 2014.

[41] Graphcore, "Intelligence Processing Unit," 2017, https://cdn2.hubspot.net/hubfs/729091/NIPS2017/NIPS\%2017\%20-\%20IPU.pdf.

[42] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz, "Understanding Sources of Inefficiency in General-Purpose Chips," in *Proceedings of ISCA*, 2010.

[43] A. Harlap, D. Narayanan, A. Phanishayee, V. Seshadri, N. Devanur, G. Ganger, and P. Gibbons, "PipeDream: Fast and Efficient Pipeline Parallel DNN Training," *arXiv preprint arXiv:1806.03377*, 2018.

[44] K. He, X. Zhang, S. Ren, and J. Sun, "Delving Deep into Rectifiers: Surpassing Human-Level Performance on Imagenet Classification," in *Proceedings of ICCV*, 2015.

[45] ——, "Deep Residual Learning for Image Recognition," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[46] A. Hodgkin and A. Huxley, "A Quantitative Description of Membrane Current and its Application to Conduction and Excitation in Nerve," *Journal of Physiology*, no. 117(4), 1952.

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

[48] Y. Huang, Y. Cheng, A. Bapna, O. Firat, M. X. Chen, D. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu *et al.*, "Gpipe: Efficient Training of Giant Neural Networks Using Pipeline Parallelism," *arXiv preprint arXiv:1811.06965*, 2018.

[49] D. H. Hubel and T. N. Wiesel, "Receptive fields, binocular interaction and functional architecture in the cat's visual cortex," *The Journal of physiology*, vol. 160, no. 1, pp. 106–154, 1962.

[50] E. Hunsberger and C. Eliasmith, "Spiking Deep Networks with LIF Neurons," 2015, arXiv preprint 1510.08829.

[51] ——, "Training Spiking Deep Networks for Neuromorphic Hardware," in *arXiv preprint arXiv:1611.05141*, 2016.

[52] A. Jain, A. Phanishayee, J. Mars, L. Tang, and G. Pekhimenko, "Gist: Efficient Data Encoding for Deep Neural Network Training," in *Proceedings of ISCA*, 2018.

[53] Z. Jia, S. Lin, C. R. Qi, and A. Aiken, "Exploring Hidden Dimensions in Accelerating Convolutional Neural Networks," PMLR, pp. 2274–2283, 2018.

[54] Z. Jia, M. Zaharia, and A. Aiken, "Beyond Data and Model Parallelism for Deep Neural Networks," *arXiv preprint arXiv:1807.05358*, 2018.

[55] A. Joubert, B. Belhadj, O. Temam, and R. Héliot, "Hardware Spiking Neurons Design: Analog or Digital?" in *Proceedings of IJCNN*, 2012.

[56] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani,

C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-Datacenter Performance Analysis of a Tensor Processing Unit," in *Proceedings of ISCA-44*, 2017.

[57] P. Judd, J. Albericio, T. Hetherington, T. Aamodt, N. Jerger, R. Urtasun, and A. Moshovos, "Reduced-Precision Strategies for Bounded Memory in Deep Neural Nets," 2016, arXiv preprint 1511.05236v4.

[58] A. Kahng, B. Li, L.-S. Peh, and K. Samadi, "ORION 2.0: A Fast and Accurate NoC Power and Area Model for Early-Stage Design Space Exploration," in *Proceedings of DATE*, 2009.

[59] S.-C. Kao, G. Jeong, and T. Krishna, "Confuciux: Autonomous Hardware Resource Assignment for DNN Accelerators Using Reinforcement Learning," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020.

[60] S.-C. Kao and T. Krishna, "GAMMA: Automating the HW Mapping of DNN Models on Accelerators via Genetic Algorithm," in *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE, 2020.

[61] L. Khacef, N. Abderrahmane, and B. Miramond, "Confronting machine-learning with neuroscience for neuromorphic architectures design," in *2018 International Joint Conference on Neural Networks (IJCNN)*, 2018.

[62] M. M. Khan, D. R. Lester, L. A. Plana, A. Rast, X. Jin, E. Painkras, and S. B. Furber, "SpiNNaker: Mapping Neural Networks onto a Massively-Parallel Chip Multiprocessor," in *Proceedings of IJCNN*, 2008.

[63] S. Kheradpisheh, M. Ganjtabesh, S. Thrope, and T. Masquelier, "STDP-based spiking deep neural networks for object recognition," *arXiv preprint arXiv:1611.01421v1*, 2016.

[64] D. Kim, J. H. Kung, S. Chai, S. Yalamanchili, and S. Mukhopadhyay, "Neurocube: A Programmable Digital Neuromorphic Architecture with High-Density 3D Memory," in *Proceedings of ISCA-43*, 2016.

[65] J. Kirkpatrick, R. Pascanu, and e. a. Neil Rabinowitz, "Overcoming Catastrophic Forgetting in Neural Netowkrs," in *Proceedings of national academy of sciences 114.13*, 2017.

[66] A. Kosson, V. Chiley, A. Venigalla, J. Hestness, and U. Köster, "Pipelined Backpropagation at Scale: Training Large Models without Batches," *arXiv preprint arXiv:2003.11666*, 2020.

[67] A. Krizhevsky, "One Weird Trick for Parallelizing Convolutional Neural Networks," *arXiv preprint arXiv:1404.5997*, 2014.

[68] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," in *Proceedings of NIPS*, 2012.

[69] L. Kull, T. Toifl, M. Schmatz, P. A. Francese, C. Menolfi, M. Brandli, M. Kossel, T. Morf, T. M. Andersen, and Y. Leblebici, "A 3.1 mW 8b 1.2 GS/s Single-Channel Asynchronous SAR ADC with Alternate Comparators for Enhanced Speed in 32 nm Digital SOI CMOS," *Journal of Solid-State Circuits*, 2013.

[70] H. Kwon, P. Chatarasi, V. Sarkar, T. Krishna, M. Pellauer, and A. Parashar, "Maestro: A Data-centric Approach to Understand Reuse, Rerformance, and Hardware Cost of DNN Mappings," *IEEE micro*, vol. 40, no. 3, pp. 20–29, 2020.

[71] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based Learning Applied to Document Recognition," *Proceedings of the IEEE*, 1998.

[72] C. Lee, P. Panda, G. Srinivasan, and K. Roy, "Training Deep Spiking Convolutional Neural Networks with STDP-Based Unsupervised Pre-Training Followed by Supervised Fine-Tuning," *Frontiers in Neuroscience*, vol. 12, 2018.

[73] C. Lee, S. S. Sarwar, and K. Roy, "Enabling Spike-based Backpropagation in State-of-the-art Deep Neural Network Architectures," *arXiv preprint arXiv:1903.06379*, 2019.

[74] D. Lee, G. Lee, D. Kwon, S. Lee, Y. Kim, and J. Kim, "Flexon: A Flexible Digital Neuron for Efficient Spiking Neural Network Simulations," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*. IEEE Press, 2018, pp. 275–288.

[75] S.-W. Lee, J.-H. Kim, and e. a. Jaehyun Jun, "Overcoming Catastrophic Forgetting by Incremental Moment Matching," in *Advances in neural information processing systems*, 2017.

[76] Y. Lin, S. Han, H. Mao, Y. Wang, and W. J. Dally, "Deep Gradient Compression: Reducing the Communication Bandwidth for Distributed Training," *arXiv preprint arXiv:1712.01887*, 2017.

[77] B. Liu, Y. Chen, B. Wysocki, and T. Huang, "Reconfigurable Neuromorphic Computing System with Memristor-Based Synapse Design," *Neural Processing Letters*, no. 41(2), 2015.

[78] C. Liu, B. Yan, C. Yang, L. Song, Z. Li, and B. Liu, "A Spiking Neuromorphic Design with Resistive Crossbar," in *Proceedings of DAC*, 2015.

[79] J. Liu, H. Zhao, M. Ogleari, D. Li, and J. Zhao, "Processing-in-Memory for Energy-Efficient Neural Network Training: A Heterogeneous Approach," in *Proceedings of MICRO*, 2018.

[80] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A Robustly Optimized Bert Pretraining Approach," *arXiv preprint arXiv:1907.11692*, 2019.

[81] W. Lu, G. Yan, J. Li, S. Gong, Y. Han, and X. Li, "FlexFlow: A Flexible Dataflow Accelerator Architecture for Convolutional Neural Networks," in *Proceedings of HPCA-23*, 2017.

[82] M. McCloskey and N. J. Cohen, "Catastrophic Interference in Connectionist Networks: The Sequential Learning Problem," in *Psychology of learning and motivation, Vol. 24. Academic Press*, 1989.

[83] W. McCulloch and W. Pitts, "A Logical Calculus of the Ideas Immanent in Nervous Activity," *Bulletin of Mathematical Biophysics*, no. 5, 1943.

[84] E. Menendez, D. Maduike, R. Garg, and S. Khatri, "CMOS Comparators for high-Speed and Low-Power Applications," in *Proceedings of ICCD*, 2006.

[85] P. Merolla, J. Arthur, R. Alvarez-Icaza, A. Cassidy, J. Sawada, F. Akopyan, B. Jackson, N. Imam, C. Guo, Y. Nakamura, B. Brezzo, I. Vo, S. Esser, R. Appuswamy, B. Taba, A. Amir, M. Flickner, W. Risk, R. Manohar, and D. Modha, "A Million Spiking-Neuron Integrated Circuit with a Scalable Communication Network and Interface," *Science, vol 345, no. 6197*, 2014.

[86] P. Micikevicius, S. Narang, J. Alben, G. Diamos, E. Elsen, D. Garcia, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh *et al.*, "Mixed precision training," *arXiv preprint arXiv:1710.03740*, 2017.

[87] D. Modha, "A New Architecture for Brain-Inspired Computing," 2015, keynote at HPCA.

[88] H. Mostafa, "Supervised learning based on temporal coding in spiking neural networks," *CoRR*, vol. abs/1606.08165, 2016. [Online]. Available: http://arxiv.org/abs/1606.08165

[89] N. Muralimanohar, R. Balasubramonian, and N. Jouppi, "Architecting Efficient Interconnects for Large Caches with CACTI 6.0," *IEEE Micro (Special Issue on Top Picks from Architecture Conferences)*, Jan/Feb 2008.

[90] N. Muralimanohar *et al.*, "CACTI 6.0: A Tool to Understand Large Caches," University of Utah, Tech. Rep., 2007.

[91] S. Narayanan, A. Shafiee, and R. Balasubramonian, "INXS: Bridging the Throughput and Energy Gap for Spiking Neural Networks," in *Proceedings of IJCNN*, 2017.

[92] National Science Foundation, "NSF Real-Time Machine Learning Solicitation," 2019, https://www.nsf.gov/pubs/2019/nsf19566/nsf19566.htm.

[93] M. Naumov, J. Kim, D. Mudigere, S. Sridharan, X. Wang, W. Zhao, S. Yilmaz, C. Kim, H. Yuen, M. Ozdal *et al.*, "Deep Learning Training in Facebook Data Centers: Design of Scale-up and Scale-out Systems," *arXiv preprint arXiv:2003.09518*, 2020.

[94] A. Nere, A. Hashmi, M. Lipasti, and G. Tononi, "Bridging the Semantic Gap: Emulating Biological Neuronal Behaviors with Simple Digital Neurons," in *Proceedings of HPCA-19*, 2013.

[95] M. O'Connor, N. Chatterjee, D. Lee, J. Wilson, A. Agrawal, S. Keckler, and W. Dally, "Fine-Grained DRAM: Energy-Efficient DRAM for Extreme Bandwidth Systems," in *Proceedings of MICRO*, 2017.

[96] P. Alcorn, "Hot Chips 2017: A Closer Look At Google's TPU v2," 2017, https://www.tomshardware.com/news/tpu-v2-google-machine-learning,35370.html.

[97] A. Parashar, P. Raina, Y. Shao, Y.-H. Chen, V. Ying, A. Mukkara, R. Venkatesan, B. Khailany, S. Keckler, and J. Emer, "Timeloop: A Systematic Approach to DNN Accelerator Evaluation," in *Proceedings of ISPASS*, 2019.

[98] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. Keckler, and W. Dally, "SCNN: An Accelerator for Compressed-Sparse Convolutional Neural Networks," in *Proceedings of ISCA-44*, 2017.

[99] D. Patterson, J. Gonzalez, Q. Le, C. Liang, L.-M. Munguia, D. Rothchild, D. So, M. Texier, and J. Dean, "Carbon Emissions and Large Neural Network Training," *arXiv preprint arXiv:2104.10350*, 2021.

[100] H. Pham, M. Guan, B. Zoph, Q. Le, and J. Dean, "Efficient Neural Architecture Search via Parameters Sharing," in *International Conference on Machine Learning*. PMLR, 2018, pp. 4095–4104.

[101] Qualcomm, "Introducing Qualcomm Zeroth Processors: Brain-Inspired Computing," 2013, https://www.qualcomm.com/news/onq/2013/10/10/introducing-qualcomm-zeroth-processors-brain-inspired-computing.

[102] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "GPT-2, Open AI," 2019, https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf.

[103] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines," *Acm Sigplan Notices*, vol. 48, no. 6, pp. 519–530, 2013.

[104] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks," *CoRR*, vol. abs/1603.05279, 2016.

[105] R. Ratcliff, "Connectionist Models of Recognition Memory: Constraints Imposed by Learning and Forgetting Functions," in *Psychological review 97.2*, 1990.

[106] B. Reagen, P. Whatmough, R. Adolf, S. Rama, H. Lee, S. Lee, J. M. Hernandez, Lobato, G.-Y. Wei, and D. Brooks, "Minerva: Enabling Low-Power, High-Accuracy Deep Neural Network Accelerators," in *Proceedings of ISCA-43*, 2016.

[107] B. Rueckauer, I.-A. Lungu, Y. Hu, and M. Pfeiffer, "Theory and Tools for the Conversion of Analog to Spiking Convolutional Neural Networks," *arXiv preprint arXiv:1612.04052*, 2016.

[108] R. Schwartz, J. Dodge, N. A. Smith, and O. Etzioni, "Green AI," *Communications of the ACM*, vol. 63, no. 12, pp. 54–63, 2020.

[109] A. Sengupta, Y. Ye, R. Wang, C. Liu, and K. Roy, "Going Deeper in Spiking Neural Networks: VGG and Residual Architectures," *Frontiers in Neuroscience*, vol. 13, 2019.

[110] J. Seo, B. Brezzo, Y. Liu, B. Parker, S. Esser, R. Montoye, B. Rajendran, J. Tierno, L. Chang, D. Modha, and D. Friedman, "A 45nm CMOS Neuromorphic Chip with a Scalable Architecture for Learning in Networks of Spiking Neurons," in *Proceedings of CICC*, 2011.

[111] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. Strachan, M. Hu, R. Williams, and V. Srikumar, "ISAAC: A Convolutional Neural Network Accelerator with In-Situ Analog Arithmetic in Crossbars," in *Proceedings of ISCA*, 2016.

[112] S.Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. Horowitz, and W. Dally, "EIE: Efficient Inference Engine on Compressed Deep Neural Network," in *Proceedings of ISCA*, 2016.

[113] Y. S. Shao, J. Clemons, R. Venkatesan, B. Zimmer, M. Fojtik, N. Jiang, B. Keller, A. Klinefelter, N. Pinckney, P. Raina, S. G. Tell, Y. Zhang, W. J. Dally, J. Emer, C. T. Gray, B. Khailany, and S. W. Keckler, "Simba: Scaling Deep-Learning Inference with Multi-Chip-Module-Based Architecture," in *Proceedings of MICRO*, 2019.

[114] F. Sijstermans, "The NVIDIA Deep Learning Accelerator," in *Hot Chips*, 2018.

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

[116] J. Smith, "A Roadmap for Reverse-Architecting the Brain's Neocortex," 2019, keynote at FCRC, https://iscaconf.org/isca2019/slides/JE_Smith_keynote.pdf.

[117] L. Song, J. Mao, Y. Zhuo, X. Qian, H. Li, and Y. Chen, "HyPar: Towards Hybrid Parallelism for Deep Learning Accelerator Array," in *Proceedings of HPCA*, 2019.

[118] L. Song, F. Chen, Y. Zhuo, X. Qian, H. Li, and Y. Chen, "AccPar: Tensor Partitioning for Heterogeneous Deep Learning Accelerators," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020, pp. 342–355.

[119] E. Strubell, A. Ganesh, and A. McCallum, "Energy and Policy Considerations for Deep Learning in NLP," *arXiv preprint arXiv:1906.02243*, 2019.

[120] J. Stuecheli, "POWER8 Processor," 2013, https://www.hotchips.org/wp-content/uploads/hc_archives/hc25/HC25.20-Processors1-epub/HC25.26.210-POWER-Studecheli-IBM.pdf.

[121] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going Deeper with Convolutions," *arXiv preprint arXiv:1409.4842*, 2014.

[122] T. Taha, R. Hasan, C. Yakopcic, and M. McLean, "Exploring the Design Space of Specialized Multicore Neural Processors," in *Proceedings of IJCNN*, 2013.

[123] M. Talsania and E. John, "A Comparative Analysis of Parallel Prefix Adders," in *Proceedings of the International Conference on Computer Design (CDES)*, 2013.

[124] T. Tang, L. Xia, B. Li, R. Luo, Y. Chen, Y. Wang, and H. Yang, "Spiking Neural Network with RRAM: Can We Use It for Real-World Application?" in *Proceedings of DATE*, 2015.

[125] O. Temam, "Hardware Neural Networks: From Inflated Expectations to Plateau of Productivity," 2015, keynote at FCRC.

[126] Tesla, "Tesla Autonomy Day," 2019, https://www.youtube.com/watch?v=Ucp0TTmvqOE.

[127] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention Is All You Need," *arXiv preprint arXiv:1706.03762*, 2017.

[128] S. Venkataramani, A. Ranjan, S. Avancha, A. Jagannathan, A. Raghunathan, S. Banerjee, D. Das, A. Durg, D. Nagaraj, B. Kaul, and P. Dubey, "SCALEDEEP: A Scalable Compute Architecture for Learning and Evaluating Deep Networks," in *Proceedings of ISCA-44*, 2017.

[129] L. Wan, M. Zeiler, S. Zhang, Y. LeCun, and R. Fergus, "Regularization of Neural Networks using DropConnect," *Proceedings of International Conference on Machine Learning - 30*, 2013.

[130] N. Wang, J. Choi, D. Brand, C.-Y. Chen, and K. Gopalakrishnan, "Training Deep Neural Networks with 8-bit Floating Point Numbers," *Advances in neural information processing systems*, vol. 31, 2018.

[131] S. Williams, "Brain Inspired Computing," 2016, keynote at ASPLOS.

[132] C.-J. Wu, R. Raghavendra, U. Gupta, B. Acun, N. Ardalani, K. Maeng, G. Chang, F. A. Behram, J. Huang, C. Bai *et al.*, "Sustainable AI: Environmental Implications, Challenges and Opportunities," *arXiv preprint arXiv:2111.00364*, 2021.

[133] Y. Wu, L. Deng, G. Li, J. Zhu, Y. Xie, and L. Shi, "Direct Training for Spiking Neural Networks: Faster, Larger, Better," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, 2019, pp. 1311–1318.

[134] C. Yakopcic and T. M. Taha, "Energy Efficient Perceptron Pattern Recognition using Segmented Memristor Crossbar Arrays," in *Proceedings of IJCNN*, 2013.

[135] A. Yang, "Deep Learning Training At Scale: Spring Crest Deep Learning Accelerator (Intel Nervana NNP-T)," in *2019 IEEE Hot Chips 31 Symposium*. IEEE, 2019, pp. 1–20.

[136] B. Yang, J. Zhang, J. Li, C. Ré, C. Aberger, and C. De Sa, "Pipemare: Asynchronous Pipeline Parallel DNN Training," *Proceedings of Machine Learning and Systems*, 2021.

[137] X. Yang, M. Gao, Q. Liu, J. Setter, J. Pu, A. Nayak, S. Bell, K. Cao, H. Ha, P. Raina, C. Kozyrakis, and M. Horowitz, "Interstellar: Using Halide's Scheduling Language to Analyze DNN Accelerators," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 369–383.

[138] Y. You, I. Gitman, and B. Ginsburg, "Scaling SGD Batch Size to 32k for Imagenet Training," *arXiv preprint arXiv:1708.03888*, vol. 6, p. 12, 2017.

[139] Y. You, J. Li, S. Reddi, J. Hseu, S. Kumar, S. Bhojanapalli, X. Song, J. Demmel, K. Keutzer, and C.-J. Hsieh, "Large Batch Optimization for Deep Learning: Training Bert in 76 Minutes," *arXiv preprint arXiv:1904.00962*, 2019.

[140] Y. You, Z. Zhang, C.-J. Hsieh, J. Demmel, and K. Keutzer, "Imagenet Training in Minutes," in *Proceedings of the 47th International Conference on Parallel Processing*, 2018, pp. 1–10.

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

[142] S. Zhou, Y. Wu, Z. Ni, X. Zhou, H. Wen, and Y. Zou, "DoReFa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients," *arXiv preprint arXiv:1606.06160v2*, 2016.

[143] B. Zoph and Q. V. Le, "Neural Architecture Search with Reinforcement Learning," *arXiv preprint arXiv:1611.01578*, 2016.

[144] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le, "Learning Transferable Architectures for Scalable Image Recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 8697–8710.