Ph.D. Dissertation of JEONGWOO PARK

Design of Low-Power Neural Network Training Accelerators through Highly Quantized Learning Process

양자화된 학습을 통한 저전력 딥러닝 훈련 가속기 설계

FEBRUARY 2022

Graduate School of Convergence
Science and Technology
Seoul National University
Intelligent Systems Major

JEONGWOO PARK

Design of Low-Power Neural Network Training Accelerators through Highly Quantized Learning Process

양자화된 학습을 통한 저전력 딥러닝 훈련 가속기 설계

지도교수 전동석

이 논문을 공학박사 학위논문으로 제출함

2022년 2월

서울대학교 대학원

융합과학기술대학원 지능형 융합시스템 전공

박 정 우

박정우의 공학박사 학위 논문을 인준함

2022년 2월

위 원 장: 이 재 욱 (인)

부위원장: 전 동 석 (인)

위 원: 김 장 우 (인)

위 원: 최 우 석 (인)

위 원: 정 두 석 (인)

Abstract

With the advent of the deep learning era, the computational need for processing deep neural networks (DNN) have increased dramatically, both in terms of performing training the neural networks on various tasks as well as in performing inference on the trained neural networks for specific use cases. To address those needs, many custom hardware ranging from systems based on field-programmable gate arrays (FPGA) or application-specific integrated circuits (ASIC) for deployment inside data centers to acceleration blocks in system-on-chip (SoC) for low-power processing in mobile devices were proposed. In this dissertation, custom integrated circuits hardware for energy efficient processing of training neural networks are designed, fabricated, and measured for evaluation of different methodologies that could be utilized for more energy efficient processing under same training performance constraints. In particular, these methodologies are categorized to three different categories for evaluation: (1) **Training algorithm**. While standard deep neural network training is performed with the back-propagation (BP) algorithm, we investigate various training algorithms, such as neuromorphic learning algorithms with spiking neurons or bio-plausible algorithms with asymmetric feedback for exploiting computational properties for more efficient hardware implementation. (2) **Low-precision arithmetic**. One of the most powerful methods for increased efficiency in DNN accelerators is through scaling numerical precision. While utilizing low precision numerics for inference phase of DNNs is well studied, training DNNs without performance degradation is relatively more challenging. A novel numerical scheme for training DNNs in various models and scenarios is proposed in this dissertation. (3) **System implementation techniques**. In actual realization of a custom training system in integrated circuits, nearly infinite design space leads to vastly different quality of results depending on dataflow inside the chip, system load balancing, acceleration and gating blocks, et cetera. Different design techniques which leads to better performance and efficiency are introduced in this dissertation.

First, a neuromorphic learning system for classifying handwritten digits (MNIST) is introduced. This learning system aims to deliver low training overhead while maintaining the training performance of classical machine learning. In order to achieve this goal, a neuromorphic learning algorithm is modified for lower operation count and memory buffer requirement while maintaining or even obtaining higher machine learning performance. Moreover, implementation techniques such as update skipping mechanism and lock-free parameter updates allow even lower training overhead, dynamically reducing training energy overhead from 25.6% to 7.5%. With these proposed methodologies, this system greatly improves the accuracy-energy trade-off in on-chip learning system as well as showing close learning performance to classical DNN training through back propagation.

Second, a programmable DNN training processor with a custom numerical format is introduced. While prior DNN inference accelerators have utilized 8-bit integers, implementing 8-bit numerics for a training accelerator remained to be a challenge due to higher precision requirements in the backward step of DNN training. To overcome this limitation, a custom 8-bit floating point format dubbed 8-bit floating point with shared exponent bias (FP8-SEB) is introduced in this dissertation. Moreover, a processing architecture of 24-way fused-multiply-adder (FMA) tree greatly increases processing energy efficiency per MAC, while complemented with a novel 2-dimensional routing data-path for making use of spatiality to increase data reuse in both forward, backward, and weight gradient step of convolutional neural networks. This DNN training processor is implemented with a custom vector processing unit, acceleration instructions, and DMA in external DRAMs for end-to-end DNN training in various models and datasets. Compared against prior low-precision training processor in ResNet-18 training, this work achieves 2.48× higher energy efficiency, 43% less DRAM accesses, and 0.8%p higher training accuracy.

Both of the designs introduced are fabricated in real silicon and verified both in simulations and in physical measurements. Design methodologies are carefully evaluated using simulations of the fabricated chip and measurements with monitored data and power consumption under varying conditions that expose the design techniques in effect. The efficiency of various biologically plausible algorithms, novel numerical formats, and system implementation techniques are analyzed in discussed in this dissertations based on the obtained measurements.

keywords: Neural Network Accelerator, Digital Integrated Circuits, Very Large Scale Integration, Computational Efficiency, Learning Systems, Neuromorphics, Machine Learning Accelerators

student number: 2017-22051

Contents

List of Tables

List of Figures

Chapter 1

Introduction

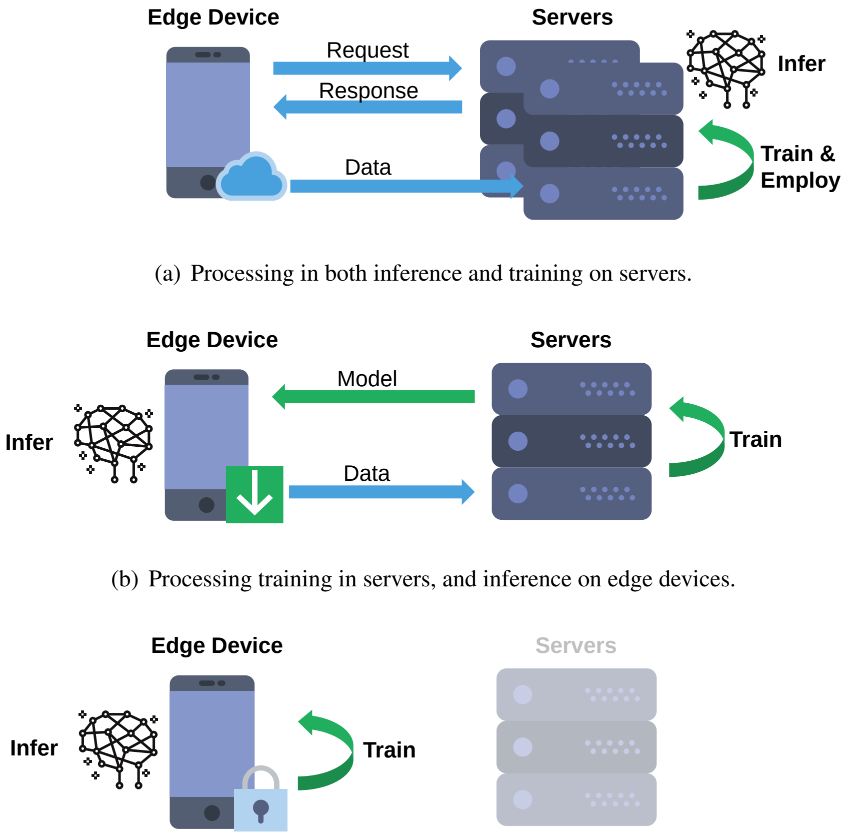
1.1 Study Background

The recent advances in deep learning algorithms have driven the need for specialized hardware systems that process compute-intensive deep neural networks (DNNs) in an energy-efficient manner. Accordingly, many research and industrial application-specific integrated circuits (ASIC) and processing units designed for neural network acceleration have been proposed and deployed for efficient processing [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] . Generally, processing of DNNs consist of two different types of processing phases. The first type is called inference, where a deep learning model is used on real data from end users where DNN models are used for performing practical tasks such as image detection, generating super-resolution images, translating sentences to other languages, etc. The other type of processing is called training, or learning, where DNN models' parameters are fit and adjusted according to some learning rule, based on showing filtered data, sometimes with labeled data and at other times without labels to the DNN model.

While both training and inference of DNN models were first processed inside data centers, using graphical processing units (GPUs) and sometimes with customized hardware such as tensor processing units (TPUs) [10], other processing schemes with inference near the edge device of the end user were also being adapted. Works such as [7, 38] were part of SoC systems inside mobile devices, intended for inference of DNN models. These processing styles have the advantage that this processing style 1) shows less latency due to routing between edge device and data centers being avoided, 2) does not require network connection for performing inference, and 3) could avoid private data being sent to data centers.

While these two processing styles are categorized by the place where inference operations occur, another interesting processing scheme is performing training on edge devices, as shown in Figure 1.1. The first advantage of training on the edge devices is that neural networks customized for each end user such as customized next-word prediction could be served without violating one's privacy. Moreover, a training scheme called federated learning [12] could be deployed. Federated learning is a novel processing training scheme where the end user's devices are used to calculate the gradient values of parameters in a model based on the data on that device. Only these gradients, not the private data, are aggregated to the main server sporadically to update the DNN model for training. However, in order to realize edge-device training, an energy-efficient method for DNN training must first be satisfied.

One of the most reliable and concrete method for building more energy efficient DNN processor is through scaling numerical precision in the processing models, exploiting the numerical robustness of DNN models. For inference phase of DNN models, use of 8-bit integers with 32-bit accumulators became one of the standard practices [13] while some extreme works [14] reduced down to using only single bit representations for both weights and activations for inference, although this method suffered from some loss of machine learning task performance. However, the numerical robustness displayed during inference phase of DNN models could not be said the same for training the same models. When training DNN models with conventional back-propagation (BP) algorithm, a pioneering work [15] showed that training neural networks even for a simple machine learning task such as MNIST digit classification showed accuracy



(a) Processing in both inference and training on servers.

(b) Processing training in servers, and inference on edge devices.

(c) Processing both training and inference on the edge devices.

Figure 1.1: Three different processing schemes for DNN models.

degradation when gradients, weights and activations were quantized to 8-bit fixed point numbers, although some compensation techniques such as stochastic rounding [15] could be deployed for better training convergence. More recent works [16] demonstrated optimization methods which provide better results using fixed-point arithmetic on larger models, but still suffer from machine learning task performance degradation compared against models trained with full precision. Therefore, it could be concluded that larger arithmetic units and data transmissions are typically required for training a DNN model compared against inference for the same model.

Moreover, the changing environments in DNN models deployed for training and

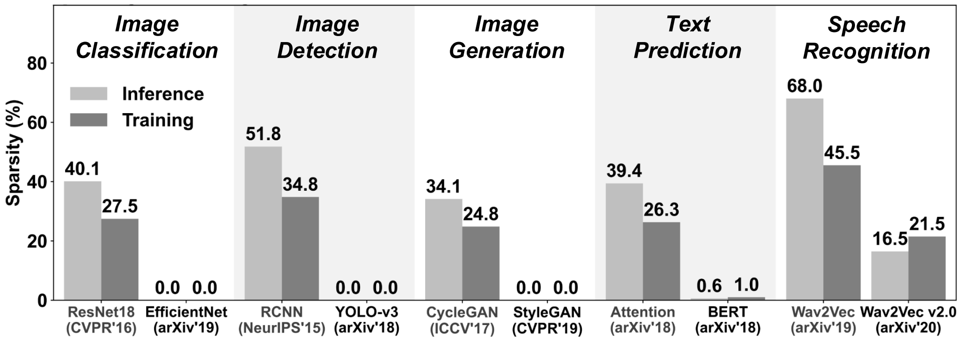
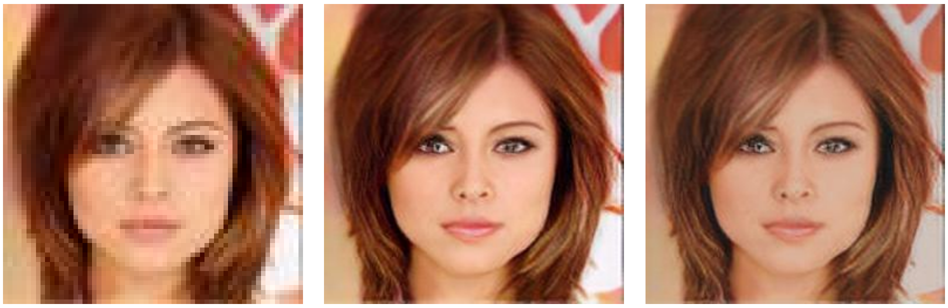


Figure 1.2: Sparsity of state-of-art deep learning models have declined significantly compared to their predecessors.

inference also calls for different designs that meets the challenges that new models face. For example, many of the prior works that builds either inference or training DNN hardware were designed under the assumption that 8-bit integers provide enough precision for inference without quality degradation, and that the ReLU (**Re**ctified **L**inear **U**nit) activation function commonly used provide ample *sparsity* in the activation maps. However, both of these assumptions have quickly changed in more recent state-of-art DNN models. For example, the deep learning community has come to favor other nonlinear activation functions such as Leaky Relu or Swish [35, 11]. This was due to some of the weaknesses of the traditional ReLU function such as gradient underflow for generative models or being weak against adversarial gradient attacks [11]. As such, the analysis of the change in sparsity of activation maps in many state-of-art models in different machine learning benchmarks shows a decline in average sparsity in activation maps compared to their predecessors. We analyzed state-of-art models in image classification, image generation, image recognition, language modeling, and speech recognition, shown in Figure 1.2. This figure implies that future DNN processors could not rely on sparsity for gaining more energy efficiency out of standard DNN models deployed, requiring a design that could perform efficiently not just on models with sparse activations, but also perform reliably on DNN models with dense activation maps.



(a) Low resolution image. (b) Enhanced with full precision. (c) Enhanced with 8-bit integer.

Figure 1.3: More error-sensitive tasks may result in quality of results degradation in 8-bit fixed point math.

Moreover, the numerical precision requirements of *inference* are getting higher. DNN accelerators must account to not just image classification models, but hold robust numerical precision for more error-sensitive models such as generative models. To illustrate this point, a low-resolution image shown in Figure 1.3(a) is enhanced with two different versions (model with full-precision arithmetic and a model with 8-bit integer arithmetic) of a super-resolution generative model, ESRGAN [47] with 23-layer residual-in-residual block architecture. The full-precision model showed a PSNR of 27.79dB compared to the original image, with the generated result shown in Figure 1.3(b). The 8-bit integer model showed a PSNR of 22.43dB compared to the original image, with the generated result shown in Figure 1.3(c). In summary, our analysis on a super-resolution task showed a PSNR degradation of 5.56dB, not only with quantitative degradation but with qualitative degradation as well, displaying clear artefacts affecting the quality of generated image.

These environmental changes offer new challenges faced by mobile deep learning processors; they must process non-sparse networks efficiently, maintain higher precision for more challenging tasks, and provide features for training neural networks with minimal hardware overhead.

1.2 Purpose of Research

As discussed in section 1.1, the drive for building low-power DNN training system has increased, while numerical scaling of DNN training proves to be more error-prone compared to its inference operations. In this background, this research proposes new methods for building more energy efficient DNN training systems while maintaining the same level of accuracy. Moreover, these methods are verified and evaluated in real integrated circuits systems that were taped out and verified on a silicon level in real time to show concrete evidence for the validity of the newly proposed design techniques.

In more detail, the proposed design methods could be categorized into three different areas of optimization. (1) First category is the training algorithm itself, where some learning rules such as neuromorphic algorithms that are considered to be more energy efficient compared against back-propagation based learning rules in conventional DNN training [18]. (2) The next area of optimization is in the precision scaling. This work proposes a novel floating point arithmetic that has distinct characteristics from standard IEEE floating point, designed specifically for training neural networks on various machine learning tasks. (3) Lastly, design techniques for digital circuits that could exploit properties of the learning algorithms and the numerical precision pipeline are proposed and evaluated.

The energy efficiency of the proposed design techniques in these three different areas of optimization are evaluated and analyzed through two real digital integrated circuits systems designed and taped out for this research. The first design is a neuromorphic learning processor [19], which is designed using a modified version of neuromorphic algorithm proposed in [20]. Works such as [25, 26, 46] have demonstrated the energy efficiency of neuromorphic algorithms in integrated circuits systems. However, they either lack enough training performance for machine learning performance on par with state-of-art DNN models [25, 26], or they are focused more only on the inference phase of neuromorphic networks and could not be utilized for training neuromorphic networks [46]. The neuromorphic processor introduced in this thesis aims to build a neuromorphic learning system that is fit for training deep learning models that are comparable to models trained with back-propagation algorithms.

The next design is a DNN training processor [21] for processing many different types of DNN models, fit for end-to-end general-purpose training. This design is built using a custom 8-bit floating point arithmetic and has been verified for real time training in selected DNN models. This DNN training processor aims to verify the efficiency of the custom 8-bit floating point, as well as proving the energy efficiency of the novel routing scheme that makes use of spatiality in all stages of DNN training for reducing data access.

Through the two designs, this thesis contributes to current research into DNN training processors in the following directions:

- Modification of existing neuromorphic algorithm for hardware implementation
- System implementation techniques such as update skipping and lock-free parameter updates
- Proposing novel 8-bit floating-point number system with shared exponent bias for DNN training
- N-way fused multiply-add trees for energy-efficient training
- Flexible routing scheme for spatial processing in tree-based processing architectures
- Extending bio-plausible learning rules to DNN training processor for hardware efficiency.

1.3 Contents

The rest of the paper is organized as follows. In chapter 2, training algorithms are discussed, including modifications that were made to an existing neuromorphic algorithm and how these modifications could benefit hardware implementation. Moreover, it is shown how training algorithms that are not based on back-propagation could be beneficial in terms of hardware efficiency. Chapter 3 gives detail on the low-precision arithmetic implemented in our DNN training processor, including experiment results conducted on various DNN models. Chapter 4 introduces the two low power learning systems that implements the research discussed in chapter 2 and 3, manufactured in custom integrated circuits design. Moreover, digital circuit design techniques in those systems for low power learning is introduced. Chapter 5 discuss the analysis results of the two systems, validating the ideas discussed in the prior chapters in simulations and measurements, as well as comparing the results to similar state-of-art integrated circuits systems. Chapter 6 discusses future research directions based on the works presented in this thesis and concludes the paper.

Chapter 2

Hardware-Friendly Learning Algorithms

2.1 Modified Learning Rule for Neuromorphic System

2.1.1 The Segregated Dendrites Algorithm

The segregated dendrites [20] algorithm is a supervised neuromorphic algorithm, originally proposed with multi-layer perceptron (MLP) architecture and demonstrated on MNIST digits classification task. The original algorithm implements bio-plausible neurons with three capacitance-coupled compartments, training with asymmetric feedback paths that connect hidden layer neurons and output neurons together through spiking channels. This algorithm achieves 96.1% MNIST test accuracy through using fixed and random feedback paths, similar to feedback alignment [23] and direct feedback alignment [24] algorithms. The Segregated Dendrites algorithm has more similarities to the direct feedback alignment algorithm, as it only contains direct feedbacks from the output neurons from the hidden neurons. This direct feedback path is desirable for hardware implementation for two main reasons: (1) processing time overhead that is required for sequentially propagating the error gradients from the topmost layer to the bottom-most layer, realizing parallel processing for more efficient processing and (2) in cases where output neurons' numbers are significantly smaller compared against hidden neurons, the connections for the feedback paths are much simpler in implementation.

We first inspect the computational modeling of the neuron's behavior in the Segregated Dendrites algorithm, which are derived from the observations made in biological neurons. The neurons used in the Segregated Dendrites algorithm is made up of three compartments, apical, basal, and somatic dendrites which each hold electrical potentials that are interdependent on each other. First, the apical dendrites accumulate feedback spikes from the output neurons, translating the spikes they receive to electrical potentials through a kernel function that smooths out the received spikes through a time decayed signal to a Post-Synaptic Potential (*PSP*). Similarly, basal dendrites accumulate feedforward spikes from previous layer neurons with the same PSP mechanism. The somatic dendrite voltage (V_S) is determined through the capacitance coupling of apical and basal dendrite voltages (V_A and V_B, respectively), as shown in equation 2.1, with coupling constants denoted by g_A, g_B, and g_L.

$$dV_s/d_t = g_L * V_s(t) + g_B * (V_B(t) - V_S(t)) + g_A * (V_A(t) - V_S(t)) \tag{2.1}$$

The determined somatic voltage is stochastically translated to spikes with a sigmoid function to interpret the voltage as probability. Unlike leaky integrate and fire neuron models, the somatic potential does not drop after firing. The illustration of the hidden neurons are shown in Figure 2.1.

The apical and basal dendrites are determined through synaptic weights and post synaptic potentials (PSPs). Each of the PSPs are calculated from the kernel function in equation 2.2 and incoming spikes that correspond to that specific synaptic connection, as shown in equation 2.3. This kernel function is also known as the *dual exponential function*.

$$k(t) = (e^{-t/\tau_L} - e^{-t/\tau_S})/(\tau_L - \tau_S) \tag{2.2}$$

$$PSP_j(t) = \sum_{k \in spikes\, of\, j} k(t_{-}t_k) \tag{2.3}$$

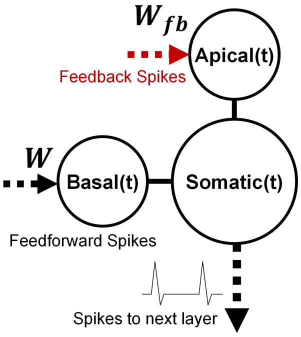
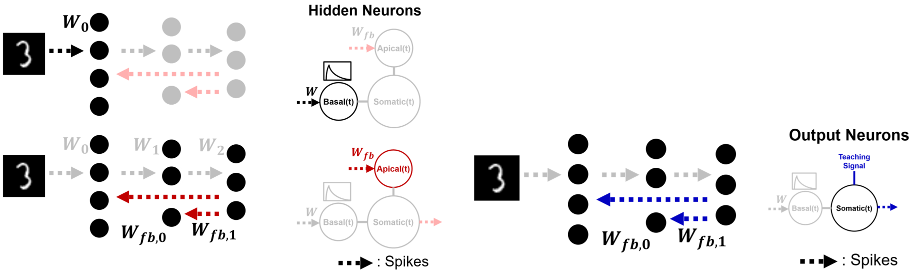


Figure 2.1: Structure of the hidden neurons in the Segregated Dendrites algorithm.



(a) Forward phase

(b) Target phase

Figure 2.2: Illustration of network behavior and interconnect in forward and target phases.

Where t_k denotes the time frame of the spike k, τ_L and τ_S each refers to long and short time constants, and j refers to the j^{th} neuron. These PSPs are translated to apical and basal dendrite potentials through equation 2.4 and equation 2.5, where W_{ff} refers to feedforward weights and W_{fb} refers to feedbackward weights.

$$V_i^A(t) = \sum_{o=0}^{M-1} PSP_o^{fb}(t) * w_{i,o}^{fb} \tag{2.4}$$

$$V_i^B(t) = \sum_{j=0}^{N-1} PSP_j^{ff}(t) * w_{i,j}^{ff} \tag{2.5}$$

In supervised training of the segregated dendrites, two phases exist to facilitate learning: the unsupervised forward phase and the supervised target phase. While different number of time steps could be allocated for the forward and target phases, the original paper suggests using 50 time steps for the forward phase and 20 time steps for the target phase. During the forward phase of training, all spikes are generated freely, with neurons' dendrite potentials determined by equations 2.1 2.5. Note that the while this phase is named 'forward', the feedback spikes are still transferred to the hidden neurons from the output neurons, making this phase *self-supervised*. The apical dendrites during the forward phase are determined through the freely generated output spikes without labels forcing output neuron behavior. In this forward phase, the average potentials are recorded to be used for weight update value calculation. However, since the nature of the algorithms are stochastic, the original paper suggests waiting some time periods to allow the network to reach a more stable plateau point. The original paper suggests waiting 20 time steps, averaging over the timesteps number 20 to 50 for generating the average potentials of apical, somatic dendrites as well as the PSPs for weight update value calculation. This forward phase behavior in the network is illustrated in Figure 2.2(a). After the forward phase, the target phase is initiated. Note that the potentials inside the neurons are not reset after a forward phase: the transition from the forward and target phase is not abrupt but a smooth process. The behavior of the neural network is the same as the forward phase everywhere except inside the output neurons, where the output neuron somatic voltage is nudged with a teaching signal to suppress all output neurons but the output neuron corresponding to the target label j. This suppression and excitation is executed through a teaching signal $I_i(t)$ given in equation 2.6.

$$I_i(t) = \begin{cases} E_E - V_{S,i}(t) & when\, i = j \\ E_I - V_{S,i}(t) & otherwise \end{cases} \tag{2.6}$$

Where E_E refers to the excitatory signal constant and E_I refers to the inhibitory signal constant. The teaching signal current I_i flows to the somatic dendrite per time step, being accumulated to the right half side of the somatic dendrite differential equation

Table 2.1: Hyperparameters used in the Segregated Dendrites Algorithm.

Notation in equations	Value	Description
g_A	0.05	Coupling conductance between apical and somatic
g_B	0.6	Coupling conductance between basal and somatic
g_L	0.3	Coupling conductance between somatic time steps
τ_L	10	Long depression constant
τ_S	3	Short potentiation constant
E_E	8	Excitatory current in target phase
E_I	-8	Inhibitory current in target phase

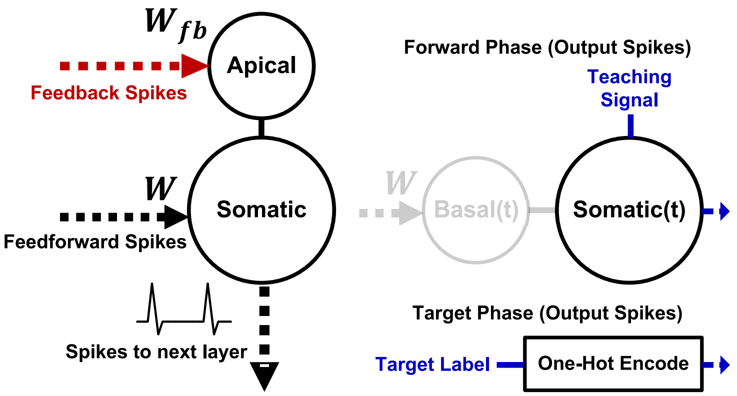
given in equation 2.1. This target phase is illustrated in Figure 2.2(b).

After the target phase, the averaged apical dendrite potential is used with the saved values from the forward phase to gain the weight update value, with the formula shown in equation 2.7. This weight update value is not added directly to the feedforward weight, but first multiplied with a learning rate hyperparameter. This learning rate differs from layer to layer in the original Segregated Dendrites algorithm. The table of hyper parameters that were described through equations that were used in this section is summarized in Table 2.1.

$$\Delta W_{ff} = \overline{PSP_{ff}} * \sigma'(\overline{V_{S,f}}) * (\sigma(\overline{V_{A,f}}) - \sigma(\overline{V_{A,t}})) \tag{2.7}$$

2.1.2 Modification of the Segregated Dendrites Algorithm

We first re-implement the CPU-based code authors provide in [20] in a GPU-accelerated framework PyTorch [49] for faster simulations of the algorithm to evaluate the training performance of any modifications to the algorithm. This re-implementation utilizes matrix-based computation in GPUs, which results in over x20 faster simulation times. Based on this GPU-based framework, we explore the original algorithm through two



(a) Modified hidden neuron model (b) Modified output neuron model

Figure 2.3: Illustration of modified hidden neuron structure. Basal dendrites are omitted, and dendrite potentials are directly calculated from spikes rather than through post synaptic potentials.

different directions: changing the neuron behavior and using different number of time steps for its impact on training performance.

For the first step in modifying the segregated dendrites algorithm for more suitable hardware implementation, we first simplify the neuron architecture to behave more similarly to point neurons in conventional deep learning. As shown in Figure 2.3, basal dendrite is omitted from determining neuron behavior and spike probability. Instead, somatic dendrite potentials are directly calculated from the input spikes and the synaptic weights. Similarly, PSPs denoted in equation 2.3 are removed from apical dendrites and apical dendrite potentials are directly calculated from feedback spikes and the feedback synaptic weights, as shown in equations 2.8 and 2.9.

$$V_S(t) = spikes_{ff}(t)W_{ff} \tag{2.8}$$

$$V_A(t) = spikes_{fb}(t)W_{fb} \tag{2.9}$$

Moreover, during the target phase, we change the teaching signal behavior from soft constraining output neuron firing patterns to directly applying one-hot encoded target label as the output neuron firing pattern, which would mean that the apical dendrite potential in equation 2.9 would change to 2.10 during the target phase.

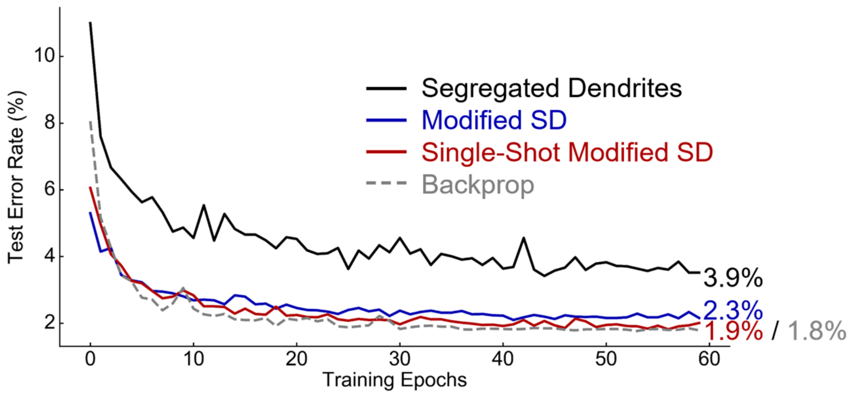
$$V_A(t) = W_{fb}[LabelIndex] \tag{2.10}$$

Inspection of the simplified neuron behavior suggests that the neuron potentials are now temporally independent and not reliant on post-synaptic potentials and differential equations. While the original SD algorithm required a minimum waiting period for neuron potentials to reach a stable operating plateau point, this new property that is observed leads us to conclude that the new neuron behavior allows waiving the constraint on the minimum waiting period due to the neurons converging quickly compared to the original algorithm. As such, shorter time periods are allowed compared against the original algorithm, and we experiment with using shorter time step training for the modified version of the segregated dendrites algorithm, even using single-time step for forward and target phases. Moreover, as the weight update formula in equation 2.7 only demand apical dendrite potential of the target phase (which are inferred deterministically from equation 2.10), we could skip target phase altogether, requiring only single-pass computation for training in this modification in the time domain. Through using single-steps for processing weight updates, we observe that the learning rule described in equation 2.7 could be simplified to equation 2.11. Although not implemented in our hardware, further optimization that saves one sigmoid operation could be made through altering the update formula to 2.12, which is an approximation of equation 2.11.

$$\Delta W_{ff} = spikes_{ff}\sigma'(V_{S,f}) \odot (\sigma(V_{A,f}) - \sigma(V_{A,t})) \tag{2.11}$$

$$\Delta W_{ff} = spikes_{ff}\sigma'(V_{S,f}) \odot \sigma(V_{A,f} - V_{A,t}) \tag{2.12}$$

The changing of the neuron model is dubbed ‘modified segregated dendrites’, and its



©2020 IEEE

Figure 2.4: Test error rate on MNIST dataset for different versions of the segregated dendrites algorithm.

single-shot implementation is dubbed ‘single-shot modified segregated dendrites’.

The test accuracy on the MNIST digit classification task is shown in Figure 2.4 for segregated dendrites, modified segregated dendrites, the single-shot modified segregated dendrites, as well as the back-propagation trained version of the same MLP architecture. While it may not be surprising that simplified neuron architectures elicit better classification accuracy, it is interesting that using single-shot results from a stochastic network reports better results compared against averaging over multiple time steps. This result is speculated to stem from the effect of increased stochasticity, which is known to yield regularization on training machine learning models [15]. Table 2.2 better supports this speculation, as it shows a trend of decreased train accuracy but increased test accuracy as time step is shortened, which is the desired effect of regularization.

Through these modifications on the original segregated dendrites algorithm, we could expect low overhead for training in terms of operations and memory required for implementing the training phase of the Segregated Dendrites algorithm. Through simplifying the neuron behavior, we no longer need to compute basal dendrites voltages to indirectly couple somatic dendrite potentials, which leads to faster convergence

Table 2.2: Training* and Test Accuracy with Varying Time Steps

Number of Time Steps	**Train Accuracy**	**Test Accuracy**
1	98.54%	98.17%
2	99.33%	98.25%
5	99.50%	97.95%
10	99.61%	98.12%
20	99.56%	97.81%

*Trained using batch size of 1

Table 2.3: Hardware Cost and MNIST Accuracy in Modified SD Algorithms

	Clock Cycles	**MNIST Accuracy**	**Training Buffer Size**	**OP/Image**
Original SD	14,000	96.1%	47.2Kb	29,379K
Modified SD	10,010	97.7%	38.4Kb	20,741K
Single-Shot Modified SD	212	98.10%	7.6Kb	804K

of somatic dendrite potentials and more even plateaus across the neurons. Moreover, the target phase could be omitted to a single time step as the output neuron behavior is deterministic. Moreover, the single-stage computations greatly reduces operation and memory requirements, reducing over ×50 as the 50 time step long forward phase is reduced to a single time step. This result is summarized in Table 2.3

In summary, the final bio-plausible algorithm that is based on modifications to the Segregated Dendrites algorithm using direct spike-only feedback requires an operational overhead of 8K accumulates and 620 sigmoid activations for computing equation 2.11, compared to its inference phase. As the inference phase of our algorithm is functionally similar to binarized DNN models, comparison against conventional BP-based learning rule gives good insight into how much overhead our training algorithm incurs. Denoting the i^{th} layer activation as a_i, i^{th} layer hidden variable before activation as h_i, and L as the evaluated loss during training, the computations required are given as equation 2.13 and 2.14.

$$\delta L/\delta W_i = a_{i-1}\sigma'(h_i)\delta L/\delta a_{i+1} \tag{2.13}$$

$$\delta L/\delta a_i = W_i\sigma'(h_i)\delta L/\delta a_{i+1} \tag{2.14}$$

Using the same configurations of the layers in our implementation (784-200-200-10 architecture), we obtain 240.8K MAC operations and 400 sigmoid activations for obtaining the required weight update values. If binarization of the activations and gradients are applied, similar to our spike-based learning rule, the MAC operation in equation 2.13 could be removed and further reduces to 42K operations and 400 sigmoid activations, which is still over 80% reduction in terms of the number of MAC counts. Therefore, it could be concluded that this modified algorithm that only utilizes feedback spikes for training exhibit lower computational requirements over the back-propagation counterpart.

2.2 Non-BP Learning Rules on DNN Training Processor

2.2.1 Feedback Alignment and Direct Feedback Alignment

While neuromorphic learning rules that resemble more biological neuron behaviors suffers from performance degradation compared against deep-learning models on more difficult tasks such as ImageNet classification, recent research on bio-plausible learning rules such as feedback alignment and direct feedback alignment [27, 28] has shown that it could scale with moderate performance degradation in modern deep learning models for ImageNet classification. These bio-plausible learning rules, unlike neuromorphic algorithms with a biological neuron behavior, utilizes point neurons similar to conventional deep learning while using a different learning rule that is not based on back-propagation. As illustrated in Figure 2.5, conventional back-propagation propagate errors through equation 2.15, where W refers to weights, e refers to error gradients, f' refers to derivative of activation nonlinear function f, a refers to activation, and i refers to the layer number.

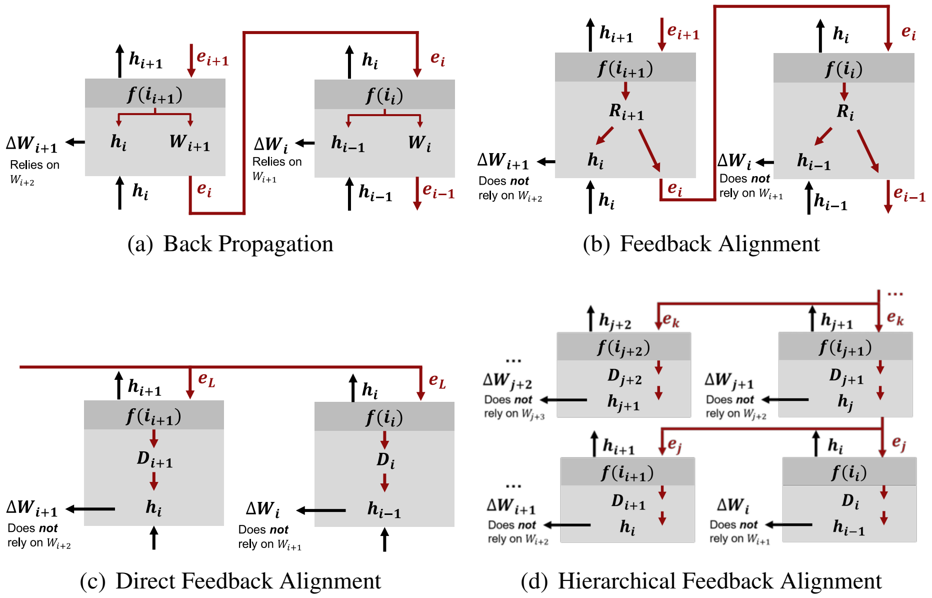
$$e_i = W_{i+1,i}^T \odot e_{i+1} f'(a_{i+1}), \quad i \in \{1, 2, 3, \ldots, L-1\} \tag{2.15}$$

This error gradient that are propagated to different layers of the neural networks are used in turn with *weight gradient* phase to generated final weight gradients. Feedback alignment (FA) algorithm and direct feedback alignment (DFA) algorithm are similar in this approach that the error gradients that are computed are used with *weight gradient* algorithms that compute final weight gradients. However, the main difference between the bio-similar algorithms (FA and DFA) and back-propagation is in the method in which the *error gradients* are computed. Feedback alignment computes the error gradients through propagate errors, similar to back-propagation, but is different in the feedback matrix that is used for next-layer error gradient computation. The feedback alignment algorithm is shown in equation 2.16. This error gradient computation path is the same except for the feedback matrix R, whereas back-propagation in equation 2.15 utilizes the weight W that was used for feedforward computation is used. Note that the R is not updated after random initialization.

$$e_i = R_{i+1,i}^T \odot e_{i+1} f'(a_{i+1}) \tag{2.16}$$

Direct feedback alignment (DFA) is deviates more from back-propagation in comparison with FA algorithm. Instead of using propagating error gradients from the prior layer for computation, error gradients in each of the layers are directly computed from the top layer error e_L and a direct feedback matrix D as shown in equation 2.17.

$$e_i = D_i^T \odot e_L f'(a_{i+1}) \tag{2.17}$$



(a) Back Propagation

(b) Feedback Alignment

(c) Direct Feedback Alignment

(d) Hierarchical Feedback Alignment

Figure 2.5: Illustration of how errors are calculated in different learning algorithms.

The motivation for developing FA and DFA algorithms is to solve the *credit assignment* problem. In conventional back-propagation in equation 2.15, the feedforward weight $W_{i+1,i}$ is used for propagating the error from layer $i + 1$ to layer i. In a biological brain, this would mean that a feedforward synaptic connection from a neuron in layer i to layer $i+1$ is also used to transmit feedback signals from that same pair of neurons in layer $i + 1$ to layer i. The problem lies in the fact that biological synaptic connections are uni-directional: in other words, back propagation could not physically be the method that our brains *learn*. However, in FA and DFA algorithms, learning is enabled without symmetric feedback and feedforward synaptic connections. Through the use of a separate feedback pathway, the learning is enabled while solving the credit assignment problem, which in turn would mean these algorithms could act as a candidate for how the real biological brain enables learning.

In terms of computational characteristics of the FA and DFA algorithms, we notice that the feedback weights R and D are generated randomly at initialization and remain constant throughout training, with no update mechanism for the feedback weights.

Moreover, DFA often suffers from scaling issues especially in CNNs both in terms of training convergence and computational complexity. The original DFA implementation require each pixels in the activation map of the CNN layers to hold feedback weights to each of the topmost layer neurons, resulting in significant increase in memory usage in deeper and larger CNN models.

In addition to FA and DFA algorithms, we propose a similar algorithm that propagates errors through hierarchical feedback paths, as shown in Figure 2.5(d). Error gradient is propagated in a hierarchical manner, where an error e_j in a local head layer is shared between the local groups with direct feedback matrix D through equation 2.18, and we hence call this error path Hierarchical Feedback Alignment algorithm (HFA).

$$e_i = D_{i,j}^T \odot e_j f'(a_{i+1}), \quad i \in \{Layers\, Connected\, to\, Local\, Head\, j\} \tag{2.18}$$

The idea behind the HFA algorithm is to bring the ideas in FA and DFA together. DFA could not scale to deeper CNNs and fails to converge as the error signals from the topmost layers are not as relevant to low-level features in the convolution neural networks. However, the hierarchical structure of DFA could provide merits to hardware implementation while providing a sort of bypass network for training deep layers, similar to Residual Networks [30]. Our HFA algorithm combines the merits of DFA with convergence stability of FA, providing a *hierarchical* feedback pathway by partially applying DFA to locally grouped layers, with the local head layer j serving as the topmost layer in DFA. In other words, instead of connecting to either the top layer (as in DFA) or to the very next layer (as in FA), layers are grouped hierarchically so that a layer group i j connects to a local head layer j. Using HFA, we expect to increase hardware efficiency while facilitating learning to a higher degree of convergence compared against conventional FA or DFA. Note that when using the HFA algorithm in convolutional neural networks, there exist a limitation on how layers could be grouped together. The layer groups must satisfy the condition that the tensor shape of the error

Table 2.4: Network-in-Network CIFAR-10 Training with Various Learning Rules

Learning Rule	**CIFAR-10 Test Accuracy**
Back-Propagation	87.59%
FA	82.03%
DFA	10.00%
HFA	85.90%
Mixture-HFA(0.8)&BP(0.2)	87.23%

gradient in each of the group layers must be the same as the error gradient shape of the local head layer. This condition should be met for proper mapping of gradients shown in equation 2.18.

In order to check the validity of the proposed algorithm, we trained a Network-In-Network [29] convolutional neural network on the CIFAR-10 dataset. We used the same training hyperparameters in back-propagation, feedback alignment, direct feedback alignment, and the proposed hierarchical feedback alignment to compare the test accuracy result. The results are shown in Table 2.4. The HFA algorithm outperforms feedback alignment algorithm by more than 3%p, and is degraded from back-propagation baseline by 1.7%p. While performance degradataion is still noticeable in HFA algorithm, this is still a major improvement over FA or DFA, which fails to converge to meaningful results.

An interesting method of training that could be implemented to provide viable learning performance while providing hardware efficiency is through alternating between back propagation and HFA algorithm in a single training session. For example, in 80% of the examples shown to a network in a single epoch, the network was trained using HFA rule while the remaining 20% was trained using back propagation. The end result show almost no degradation at 87.23%, whereas back propagation trained the network to accuracy of 87.59%. Through the use of the hybrid training method, one is able to exploit the hardware efficiency of HFA algorithm (discussed in the next sec-

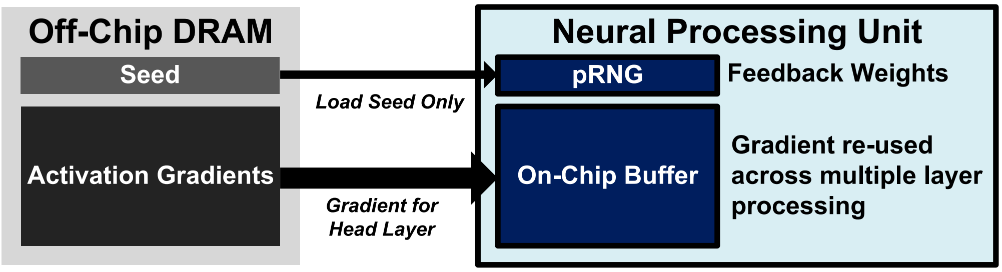


Figure 2.6: Hardware advantages of using HFA learning rules.

tion 2.2.2) while maintaining training performance. In our implementation, switching between HFA and back-propagation require very little software overhead as simply branching to different code execution address provide switching.

2.2.2 Reduced Memory Access in Non-BP Learning Rules

One of the advantages of the bio-plausible learning rules discussed in chapter 2.2.1 is that the feedback weights are generated randomly and are not updated for training. Due to this characteristic, these feedback weights could be generated on-chip with proper random seed control without accessing external memory. In the DNN training processor discussed in chapter 2.2, random number generation using LFSRs are implemented on-chip to take advantage of the bio-plausible learning rule, which allow specifying random number generation seeds with a 128-bit number initialization. Moreover, the HFA algorithm proposed in chapter 2.2.1 could be further advantageous for processing, as the gradient in the local head layer j could be reused multiple times for more efficient processing. The advantages in hardware for HFA algorithm is shown in Figure 2.6. In summary, off-chip memory access are minimized through using seed-based random weight load and head layer gradient re-use, which shows potential for minimizing memory latency and reduced power at DRAM controllers and at the DRAM itself.

Based on this observation, we create a test suite for comparing and analyzing back propagation algorithm and the two other bio-plausible learning rules (FA and HFA) discussed in section 2.2.1. This test suite is created with minimal alterations to the DNN training processor discussed in section 4.2 to better exploit the characteristics of the bio-plausible learning rules. First, a pseudo-random number generator that creates vectors of uniform distributions are implemented with minimal logic based on XOR's and LFSRs. Moreover, existing convolution backward instruction in the DNN training processor makes use of two unused flag bit fields in the instruction to (1) indicate whether to use random feedback weight generation for processing, and (2) whether to use hierarchical feedback for parallel processing of multiple layers. The control FSM is also modified to skip weight feedback loading, and instead points to LFSRs for generating weights. Last, the memory address generator for storing result gradients from convolution backward instruction for hierarchical feedback is modified for storing different tensors in parallel. These three modifications comes at a minimal cost in hardware logic, as the main processing blocks, vector processing units, and the SRAM memory that takes up more than 95% of the total power and area budget is left untouched.

With this test suite, we compared and analyzed naïve convolution layer backward instruction execution with back propagation and convolution layer backward instruction with random number generation in place of weight loading. Results that include processor energy consumption, executed with a synthesized netlist are shown in Table 2.5. Two different configurations are used – the first configuration is part of a CIFAR-10 network-in-network training with batch size of 1, while the second configuration is a layer in ImageNet ResNet-18 training [30] with a batch size of 4. The first configuration is mainly bottlenecked with loading the weight from external memory to the processor, which results in FA saving around 63.1% in total execution time and 53.3% in total energy. However, this savings are quickly dwarfed when larger batch sizes and larger images are deployed for training, as the model parameter sizes remain constant while activations and gradients are increased by $O(n)$ with increasing batch sizes. In order to test the bio-plausible algorithms in scaled environments, larger batch sizes and

Table 2.5: DNN Training Processor Latency and Energy Consumption

Configuration	Learning Rule	Latency (µs)	Energy(µJ)
128×128 channels, 8×8 features, 3×3 kernel size, Batch Size=1	BackProp	292.69	44.20
	FA	108.00	20.63
128×128 channels, 28×28 features, 3×3 kernel size, Batch Size=4	BackProp	10272.65	1980.24
	FA	9901.91	1886.86
	HFA*	6850.29	1552.48

*3 layers are attached to a local head.

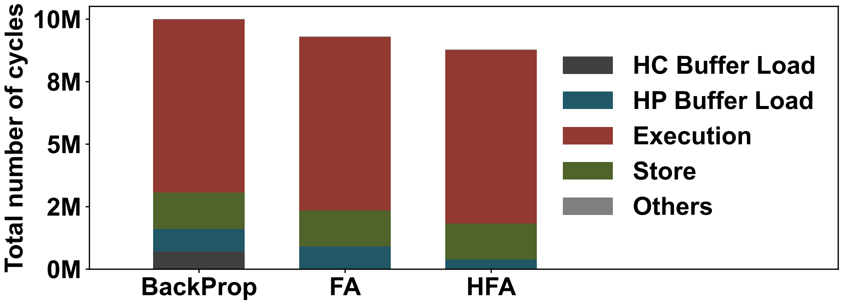


Figure 2.7: Breakdown of bio-plausible learning rule execution in ResNet-18 backward pass.

larger image feature sizes are used in the next configuration. In this configuration of using batch size of 4 on ImageNet layer configuration, memory access is more constrained by the gradient rather than the weight due to large image sizes and a larger batch size. We could observe that while FA does not scale well in this configuration (as DRAM accesses are dominated by gradients), HFA still manages to reduce total execution time by 23.3% as a result of sharing gradients and increased parallelism.

Moreover, we conducted a cycle-accurate experiment with register transfer level (RTL) code of the DNN training processor with support for optimized bio-plausible rules to evaluate system level latency and performance bottlenecks in such conditions, with the result shown in Figure 2.7. For the backward pass of ResNet-18 on the ImageNet dataset with batch size of 4, using back-propagation algorithm require 10.00M cycles in ResNet-18, with 3.06M cycles (30.6% of total execution time) spent on memory stalls. In FA implementation on our DNN training processor, same ResNet-18 backward pass required 9.29M cycles, with 2.36M cycles (25.4% of total execution time) spent on memory stalls. In HFA implementation, through sharing gradients across layers with same conv-bp configuration, we observed total execution cycle of 8.77M cycles, with 1.84M cycles (21.0% of total execution time) on memory stalls. In conclusion, compared against conventional back-propagation algorithm on ResNet-18 ImageNet training, FA require 22.9% less memory stall time and 7.1% less execution time, while HFA require 39.9% less memory stall time and 12.3% less execution time.

Chapter 3

Optimal Numerical Format for DNN Training

3.1 Related Works

As discussed in section 1, scaling numerical precision for low power training proves to be more difficult compared against scaling numerical precision for inference in DNN models. Some related works [1, 31, 32, 33, 34] have addressed this problem. Work in [1] proposed using a mixed precision of 8 bits and 16 bits for representing tensors in DNN training, while maintaining the trained accuracy of full-precision based models. The method presented in [1] first represents tensors in 8-bit floating point, while leaving the elements that are out of the dynamic range of the 8-bit floating point and hence not representable with the exponents of the 8-bit floating point numbers to be zero. Instead, these out-of-range elements are represented in the complementing 16-bit floating point tensor while the elements that has been expressed as non-zero value in the 8-bit floating point tensors are zero in this complementing 16-bit floating point tensor. This method has the major drawback that it requires representing same tensors twice in terms of 8 bits and 16 bits, which they mitigate through zero-skipping features, as shown in Figure 3.1.

Work in [31] proposes using 8 bits of floating point and 16-bit accumulators for DNN training. The major contribution of the work in [31] is identifying the point

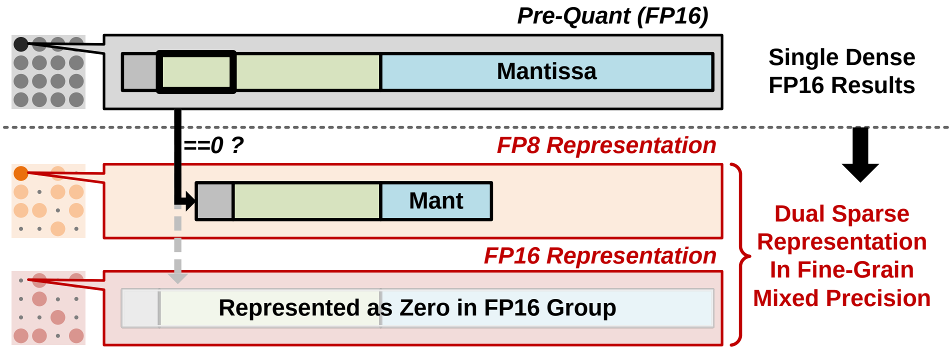

Pre-Quant (FP16)
Mantissa
Single Dense FP16 Results
==0 ?
FP8 Representation
Mant
FP16 Representation
Represented as Zero in FP16 Group
Dual Sparse Representation In Fine-Grain Mixed Precision

Figure 3.1: An illustration of the fine-grain mixed-precision training scheme introduced in [1].

where DNN training performance is lost when using 8-bit or similar low-precision floating point numeric system. It states that the major source of loss of precision is due to *swamping* effect, which is a known phenomena that occurs when two floating point numbers with one of the number having magnitudes of larger absolute value over the other are added together. When the exponent difference between the two addition operands are significantly large, precision loss is inevitable as the mantissa of the smaller number is lost in the addition and re-normalization process. This swamping effect is identified as the main source of precision loss, and the work in [31] suggests using a chunking accumulation, where groups of MAC operations are carried out before being added to the actual accumulator, in the hopes of relieving the absolute value difference between the accumulator and the multiply-added values in the dot product operations in DNN training. Moreover, other auxiliary training methods such as using 16-bit copies of the original weights and using stochastic rounding is proposed to aid DNN training in the bulk of computation carried out in 8-bit floating point numbers.

Work in [32], which succeeds the work in [31], proposed using a hybrid representation which utilizes 1.4.3 in the forward pass and 1.5.2 format during the backward and weight gradient path. (Henceforward, $1.X.Y$ refers to using 1 bit sign, X bit exponents, and Y bit mantissa) The purpose of this hybrid approach is to compensate the

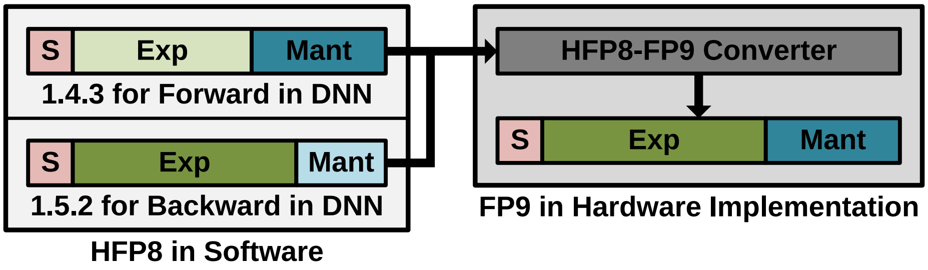
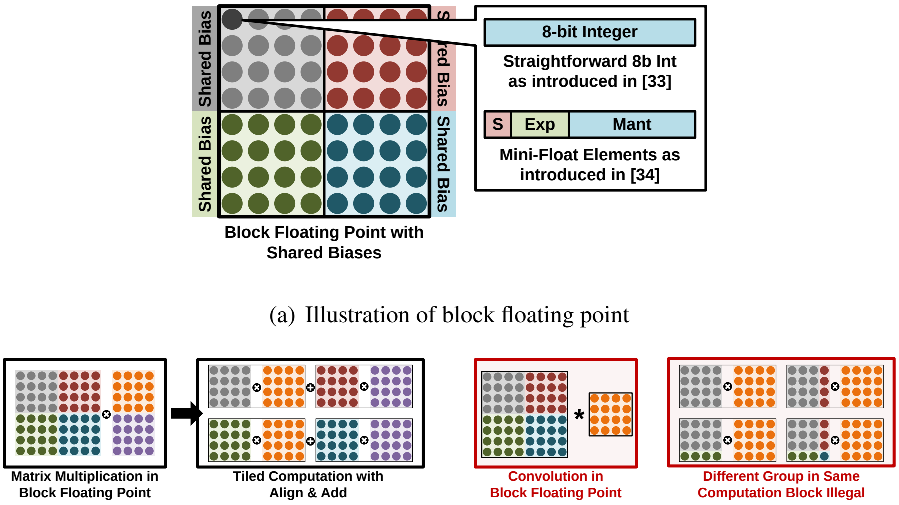


Figure 3.2: An illustration of the hybrid-fp8 arithmetic block introduced in [5].

training performance loss that is observed in some machine learning taks and models observed following the straightforward 1.4.3 quantization format used in the work of [31]. Through using different arithmetic of 1.4.3 and 1.5.2, this work demonstrates that using two different types of 8-bit precision for DNN training shows a training performance without any accuracy loss. This hybrid method incurs some additional hardware, whereas the hardware implementation work in [5]showed that this cost could be reduced through a hybrid implementation of FP9 format (1.5.3) and an FP8-FP9 converter, as shown in Figure 3.2.

Lastly, works in [33, 34] shows utilizing block floating point for DNN training. Work in [33] proposed grouping tiles of N×N elements with 10-bit exponent value, where each of the elements are represented with 8-bit mantissa. Work in [34] generalizes the work in [33] to using a mini float representation for each of the elements instead of using only mantissa in the elements, as shown in 3.3(a). This allows exponents to differ in the elements and share biases instead of exact exponent values. This strategy works well for matrix multiplication and dot products, where the cost of accumulating the tiled results including aligning different biases could be amortized over the length of the dot product, as shown in 3.3(b). However, when applied to spatial architectures for processing convolutional layers, this strategy incurs more overhead compared to simple matrix multiplication as overlapping components inside a tensor to be convolved require special handling (such as representing same element twice or


Shared Bias
Shared Bias
Shared Bias
Shared Bias
8-bit Integer
Straightforward 8b Int as introduced in [33]
S
Exp
Mant
Mini-Float Elements as introduced in [34]
Block Floating Point with Shared Biases
Matrix Multiplication in Block Floating Point
Tiled Computation with Align & Add
Convolution in Block Floating Point
Different Group in Same Computation Block Illegal

(a) Illustration of block floating point

(b) Matrix multiplication in block floating point

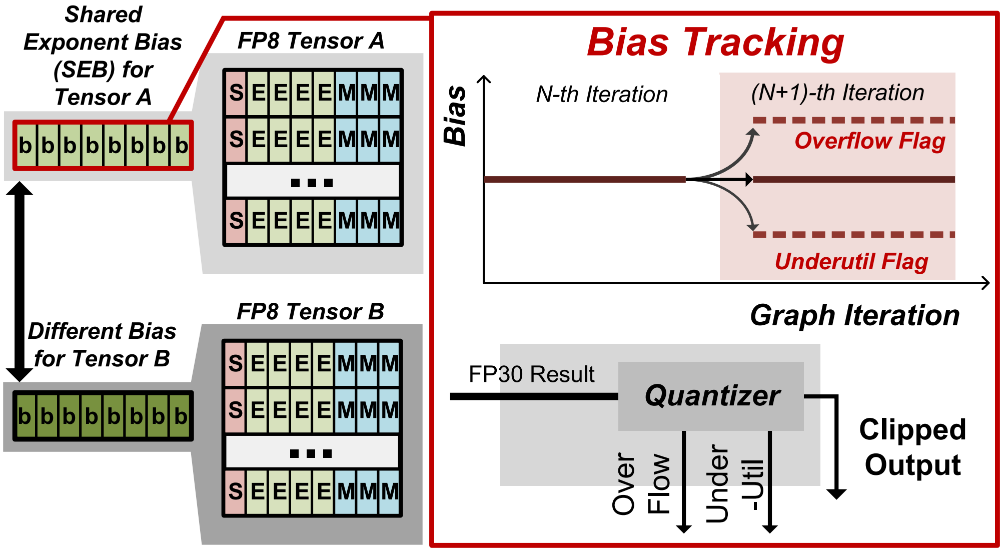
(c) Convolution in block floating point

Figure 3.3: An illustration of the dual-representation problem in block floating point.

using smaller block sizes), as shown in Figure 3.3(c). This is exemplified in an DNN accelerator [45] that adopts block floating point for training, which exhibits 78% lower TOP/s for ResNet-101 (which is consisted mostly of convolutions) compared against LSTM models (consisted mostly of matrix multiplications).

3.2 Proposed FP8 with Shared Exponent Bias

To provide a more reliable and hardware-efficient quantization method for training DNN models with 8-bit floating point numbers, this research proposes a quantization method called FP8 with Shared Exponent Bias (FP8-SEB) that does not require realignments or hybrid computing hardware that effectively produces 9-bit floating point computation units. FP8-SEB represents elements of a tensor with 1-bit sign, 4-bit exponents, and 3-bit mantissa, while sharing a common exponent bias for all the elements of the tensor. Unlike the work in [33, 34], we group all of the elements of the output activation in a layer as a tensor, weight in a layer as a single tensor, etc., instead of tiling those tensors in small groups. To compensate for the shift of dynamic range in



©2021 IEEE

Figure 3.4: Illustration of FP8-SEB. Biases are tracked during each computation pass with overflow/under-utilization flag to adjust to dynamic range changes.

tensors during progression, we automatically modify the exponent bias value through tagging each tensor with flags that indicate whether it has been overflown (where the bias values are increased by 1) or underutilized (similarly, bias values are decreased by 1). The illustration of this FP8-SEB is shown in Figure 3.4.

In this numerical format, the real value for an FP8-SEB number that is represented with bias b, 1-bit sign value s, 4-bit exponent value e, and 3-bit mantissa value m could be retrieved with equation 3.1.

$$x^Q = (-1)^s 2^{e-127+b}(1+m/8) \tag{3.1}$$

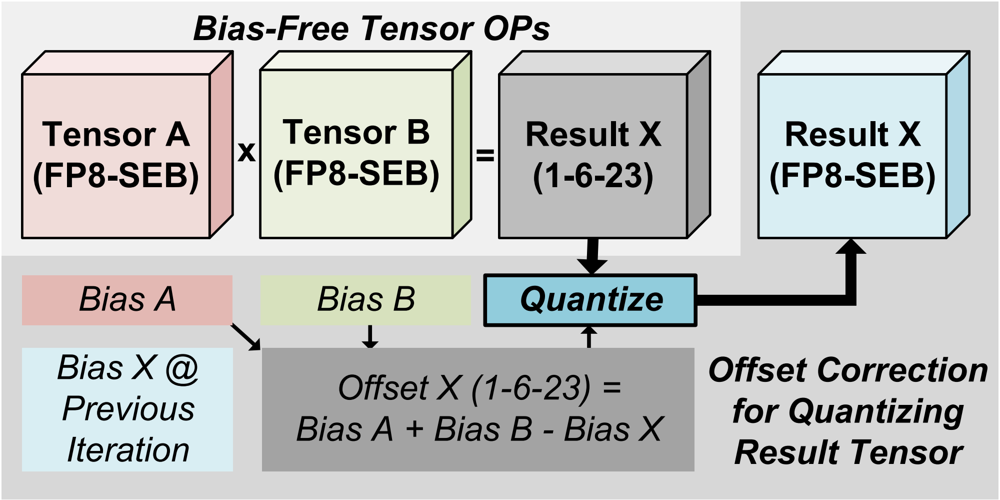
As this representation format require using different bias values on each of the tensor elements that are grouped together, the additional cost that is associated with using different bias values for each tensor should be analyzed carefully since they are an overhead that do not exist for other quantization methods. Firstly, in terms of memory cost, the cost for storing the shared exponent bias should be considered. Our analysis shows that this cost is negligible compared against the cost of actual elements of the tensor: for example, in ResNet-18 ImageNet training, the cost of shared biases, including the flag data for overflow and under-utilization, accounts for only 0.003% of the entire memory allocated for ResNet-18 training.

Next, it should be considered whether hardware implementation require extra logic to account for the fact that biases could take any value rather than being fixed. For some arithmetic operations such as addition between FP8-SEB tensors, this case is true as during normalization, realignment logic should be applied for arbitrary biases that are not known priori. However, for matrix multiplication and convolution operations, careful observations show that we scarcely require extra logic for using different biases that are not determined during computation. To provide insight into the implementation method that avoids extra logic, a dot product operation between two vectors X_1 and X_2 with length N is shown in equation 3.2. By substituting x_1 and x_2 with equation 3.1 and rearranging, we obtain equation 3.3.

$$y_{dot} = \sum_{i=0}^{n-1} x_{1,i}^{Q} x_{2,i}^{Q} \tag{3.2}$$

$$y_{dot} = 2^{b_1+b_2-254} \sum_{i=0}^{n-1} (-1)^{s_{1,i}+s_{2,i}} 2^{e_{1,i}+e_{2,i}} (1+m_{1,i}/8)(1+m_{2,i}/8) \tag{3.3}$$

Observation into equation 3.3 show that computations related to the unfixed biases could be decoupled from the bulk of dot product computation inside the summation. Thus, after bias-free operations are complete, only on the accumulated value that we need to apply re-quantization accounting for the bias values. As this biases are fixed across all tensors, this could be easily scaled to matrix-matrix multiplications or convolutions. This computation process for matrix multiplication in FP8-SEB is shown in Figure 3.5. Therefore, only hardware overhead for matrix multiplication and convolution in FP8-SEB is in re-quantization step after bulk of computation is finished. As an example, suppose that FP30 (1.6.23 format) is chosen for accumulator precision, and


Bias-Free Tensor OPs
Tensor A (FP8-SEB)
x
Tensor B (FP8-SEB)
=
Result X (1-6-23)
Result X (FP8-SEB)
Bias A
Bias B
Quantize
Bias X @ Previous Iteration
Offset X (1-6-23) = Bias A + Bias B - Bias X
Offset Correction for Quantizing Result Tensor

©2021 IEEE

Figure 3.5: Matrix multiplication in FP8-SEB. Convolution operations could be carried out similarly without re-quantization, as biases are shared across entirety of the tensor.

the result target is expressed as Y_Q with a known exponent bias $b_(Y_Q)$. Re-quantization is carried out as equation 3.4-3.6 from a final result Y_acc in FP30 format, shown in equation 3.7:

$$S_{Y_Q} = S_{Y_{acc}} \tag{3.4}$$

$$e_{Y_Q} = clip(e_{Y_{acc}} - (b_1 + b_2 - b_{Y_Q}) + 254, 0, 15) \tag{3.5}$$

$$m_{Y_Q} = round(m_{Y_{acc}/2^{20}}) \tag{3.6}$$

$$Y_{acc} = (-1)^{S_{Y_{acc}}} 2^{e_{Y_{acc}}} (1 + m_{Y_{acc}}/2^{23}) \tag{3.7}$$

3.3 Training Results with FP8-SEB

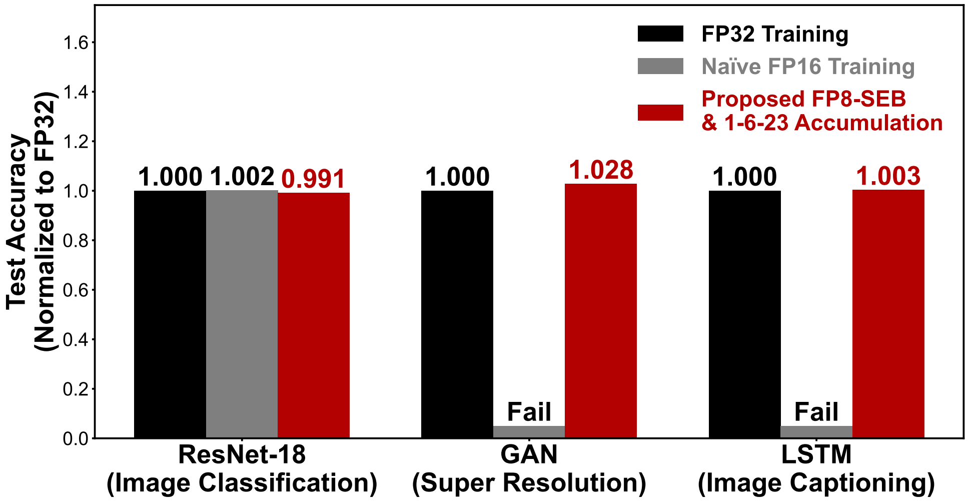
One of the advantages of using explicit separate biases in each of the tensors used during training is that they provide more inter-tensor dynamic range while maintaining intra-tensor dynamic range. Dynamic adjusting shown in Figure 3.4 makes sure that all of the dynamic range in the 4-bit exponents are utilized, while allowing different tensors such as weight gradients, weights, and activations, which have huge difference in terms of their dynamic ranges [17], to all be expressed accurately. Therefore, our numerical format correctly addresses the shift in dynamic range in tensors due to training progression.

However, the accumulation precision must be considered for processing matrix multiplication accurately without final neural network training degradation, as suggested in [31]. While FP8 tensors provide high intra-tensor dynamic range for representing unique data, expressing such large dynamic ranges with limited precision inevitably results in lower grain precision in between data-points, leading to devastating precision loss during accumulation. For instance, suppose an addition occurs between two positive FP8 elements (1'b0, E_a, M_a) and (1'b0, E_b, M_b), where the element with larger exponent value is assigned a such that $E_a \geq E_b$. The addition process including alignment is expressed as equation 3.8.

$$C_{preQ} = 2^{E_a}(1+\frac{M_a}{2^3})+2^{E_b}(1+\frac{M_b}{2^3}) = 2^{E_a}(1+\frac{M_a + 2^{E_b-E_a}M_b + 2^{3-E_a}}{2^3}) \quad (3.8)$$

Observing the right hand term of the equation 3.8 in combination of quantization equation 3.1, we could conclude that the resulting output FP8 value (1'b0, E_c, M_C) is always equal to (1'b0, E_a, M_a) when $2^{E_b-E_a}M_b + 2^{3-E_a} < 0.5$, resulting in quantization loss. This phenomena is exemplified in real DNN training experiments when such conditions are met. Note that this is condition is always met when $E_a > E_b + 4$, meaning that all addition between elements with exponent difference larger than 4 are ignored entirely. When naïvely using the same FP8-SEB representation for intermediary accumulation during matrix multiplication operations, experiments showed 5%p accuracy degradation for even simple tasks such as LeNet training on CIFAR-10 and failed to converge at all on deeper networks such as ResNet-18.

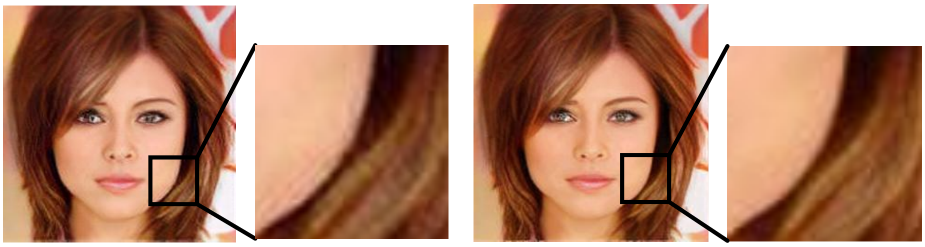
Thus, it is required to choose a higher precision format for accumulation without



©2021 IEEE

Figure 3.6: Comparison of trained accuracy on selected tasks. Half-precision training, shown in gray, fails for GAN and LSTM task while FP8-SEB with FP30 accumulators perform on par with models trained with full precision.

accuracy degradation in DNN training: while other works [31, 32] chose FP16 as accumulators, a custom FP30 (1.6.23) format was chosen for accumulation precision. This decision was based on two different reasons: (1) First reason was to bit-match the GPU simulation results. Unlike work in [31, 32], FP8-SEB lacked software modeling for changing the accumulator precision in GPU-based CUDA convolution kernels, and we instead relied on using off-the-shelf CUDA convolution kernels provided by PyTorch library and instead put quantization functions that works with . This meant that while inputs and outputs of the convolution kernels were quantized in FP8-SEB, the actual accumulators were in FP32 (1.8.23) precision. To ensure bit match with our software simulations, FP30 (1.6.23) was chosen to ensure mantissa bit preservation while reducing unnecessary exponent bits. (2) Second reason for FP30 implementation was that naïve half-precision (1.5.10) showed training failure in many cases. Figure 3.6 shows that FP8-SEB with FP30 accumulators match or even outperforms FP32 training baseline on difficult tasks such as generative task and natural language

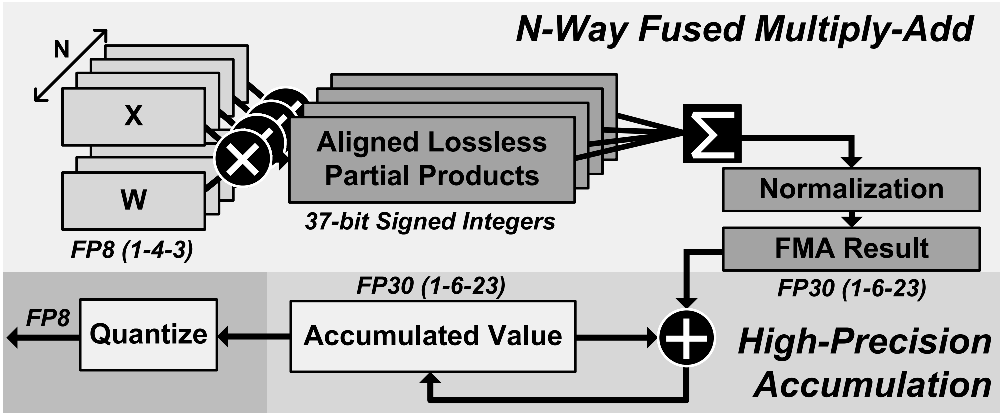


(a) GAN trained with FP32. (b) GAN trained with FP8-SEB.

Figure 3.7: Generated super-resolution images, trained in FP32 (shown on left) and FP8-SEB (shown on right).

processing (image captioning), while half-precision fails to converge for those tasks. Moreover, our FP8-SEB combined with FP30 accumulators showed even higher performance compared against the full-precision in GAN tasks consistently. We speculate this result stems from the implicit unbiased noise generated by FP8 quantization provide robust training that stochasticity provides. While quantitative results showed 3% higher results for FP8-SEB, qualitative evaluation is equally important for comparing GAN performance. To qualitatively verify the super-resolution performance of the model trained with FP8-SEB numeric format, we show the GAN-generated result on the same model, task as shown in Figure 1.3 in Figure 3.7. While super-resolution image generated from FP32 training (shown in Figure 3.7(a)) produces erratic artifacts around the edge, image generated from our GAN model trained with FP8-SEB (shown in Figure 3.7(b)) produces smoother edge lines and no irregular artifacts.

While this choice of high-precision accumulators guarantees robust training performance in various machine learning models with bit-matched simulations, this numerical format incurs a huge cost in terms of hardware implementation when they are designed straightforwardly with standard MAC-based processing in DNN processors. In the next section, we discuss the hardware and software co-optimized design method that could be utilized to mitigate the overhead incurred by the large accumulator.



©2021 IEEE

Figure 3.8: FMA trees implementation. Partial products are added together in lossless dimensions of 37-bit integers.

3.4 Fused Multiply Adder Tree for FP8-SEB

To co-optimize the numerical pipeline for matrix multiplication using FP8-SEB format and FP30 accumulators in hardware, we propose using tree-based fused multiply add operation as the basic unit for computation in our DNN training processor [21]. In this tree-based implementation, two 24 element vectors are multiplied and added together in lossless fused multiply add operation in 3 pipeline stages, with the final result re-quantized to 1-6-23 FP30 format. Figure 3.8 shows the detailed implementation in block diagrams of the hardware. In the bias-free approach discussed in section 3.2, the two input vectors are expanded and multiplied for 24 partial products. To prevent any precision loss, the partial products of the two 8-bit floating point numbers are translated to 37-bit signed integers accounting for the combined exponential range (with each element having exponent range of 0 to 15, the seven-bit multiplier result from mantissa are shifted by minimum of 0 to maximum of 30 bits, resulting in 37-bit signed integer space) for lossless conversion and added together through 24-way adder trees. After summation is complete, the 42-bit signed integer is normalized to FP30 in 1.6.23 format without bias to be added to the accumulator. This 1.6.23 format is chosen based on observation of the accumulator, choosing the minimum number of

Table 3.1: Energy Consumption & Area Per MAC, Synthesized Results

	1-Way MAC, FP8	24-Way FMA, FP8
Comb. (pJ)	5.8	1.5
Seq. & Memory (pJ)	10.1	0.4
Area*(μm^2)	3081.6	848.9

*Excludes SRAM

exponent bits that does not overflow, which was 6 bits. 23 bits of mantissa are chosen for bit-match with the IEEE full precision, which also contain 23 bits of mantissa. In order to prevent glitches from incurring large currents in long cascaded combinational logic, the FMA tree is split to 3 stages, ensuring a theoretical operating speed of less than 2 ns in 40nm LPCMOS technology in synthesis results.

There are two advantages for using tree-based computation as the basic computing unit. Firstly, the average cost for implementing the actual processing logic is reduced. The obvious reason behind is that FP30 accumulators are very costly compared against 8-bit FMAs and sharing FP30 accumulators will greatly reduce this accumulator cost. Yet another logic reduction for tree processing is due to implementation of the adder: N-way adder trees provide more space for logic optimization than parallel implementation of adders. In our synthesis results, 24-Way 37-bit integer adder tree cost 9% less in terms of combinational logic compared against 24 adders in 37-bit integers. Combined together, the combinational logic's energy cost per MAC operation is reduced from 5.8pJ in a straightforward MAC-based implementation to 1.5pJ in a 24-way fused-multiply-adder tree based implementation. Moreover, the implicit cost of accessing memory for accumulators could also be reduced. While the cost of accessing local scratchpad for loading and storing partial results vary with scratchpad sizes, the cost of accessing SRAMs or register files are still large for either the cost associated with signal amplification (in SRAMs) or with the cost associated with encoding and decoding address to data logic (in register files). For instance, in the local scratch-

Table 3.2: PSNR in Matrix Multiplication with Varying N-Way FMAs

N	1	2	4	8	16	32	64
Avg. E_x	5.26	6.72	7.97	9.03	10.02	11.00	12.00
PSNR	14.265	21.228	24.07	24.147	24.147	24.150	24.148

pad used in our design, which were foundry-provided two-port register file with 30 bit word and 256-entrees, the sequential and memory cost associated were up to 10.1 pJ per cycle. As in a 24-way FMA tree the memory access costs are reduced by 1/24 compared against conventional MAC implementations, this sequential and memory energy is reduced to 0.4pJ. The total energy per MAC, comparing 24-way FMA trees and conventional MAC is compared in Table 3.1. The second advantage in a tree-based implementation is that the matrix multiplication results show higher numerical precision statistically when implemented in FMA tree structures. The reason behind this statistical improvement is that small normalization errors that exist in floating point arithmetic (also known as 'swamping' issues) could be mitigated through accumulating more values before quantizing and adding to FP30 accumulators. In MAC operation, when the result of multiplied values is much smaller than the accumulator, the mantissa bits of multiplied values are lost during alignment and normalization. However, in FMA trees, as more values are added together in lossless dimension before being added to the accumulator, the chance of losing mantissa bits during normalization decreases. Quantitatively, expressing the accumulator as (s, E_{Acc}, M_{Acc}) and the added operand x as (s, E_x, M_x), and letting $D = 2^{E_{Acc}-Ex}$, we could quantify error as equation 3.9.

$$Error = min(D - (M_x \mod D), M_x \mod D)/2^{E_x} \tag{3.9}$$

As $D \propto 2^{-E_x}$ for a fixed accumulator, from equation 3.9 we could find $O(\frac{1}{4^{E_x}})$ relationship between the rounding error and the exponent value of x, illustrating inverse exponential relationship between the exponent value of the addition operand to the accumulator and the swamping error. We deploy more addition between the 24 element partial products before being added to the accumulator for statistically increasing the exponent value of the addition operand. To demonstrate this point, 1024×1024 matrices initialized in a uniform distribution are multiplied with varying N in N-way FMA tree accumulation. This uniform distribution encourages a result that is not centered around a non-zero value, similar to the distributions that each neurons face in real-image for a convolutional neural network as a large number of critical neurons are biased for a well trained network on real images. The multiplied results are compared against the full-precision baseline to shown peak signal to noise ratio (PSNR) in Table 3.2. The matrices were sampled from a uniform distribution and quantized to FP8-SEB format before being multiplied, which explains the fundamental upper limit of precision (24.15 PSNR dB) in Table 3.2. We denote the average exponent value of the adder operand (x) in the accumulation, and we could observe a trend of larger exponent values (Avg. E_x) for wider number of accumulation in the FMA trees.

Chapter 4

System Implementations

4.1 Neuromorphic Learning System

In this section, the overall architecture of the system and the design techniques that are implemented on a fabricated integrated circuits chip for the bio-plausible learning algorithm that were discussed in section 2.1 is described in more detail. In this implementation, a feedforward, fully connected network with 2 hidden layers and 200 neurons are chosen. Note that this design is a proof-of-concept fixed-structure design for learning on image classification. Noticeably, a parameter lock-free update algorithm and an update skipping mechanism that exploits the sparsity of the learning rule is utilized for a training operation with low overhead.

4.1.1 Bio-Plausibility

In neuromorphic learning algorithms, learning is said to be *bio-plausible* if either the neuron model or the learning rule is bio-plausible [18]. For instance, neuron models used in some neuromorphic systems mimic biological neurons, using spikes, or 1-bit information as its inputs and outputs. Moreover, they are often modeled using differential equations in the time domain, determining their spike using the potential voltages described by the equations [20]. Examples of such biological neuron models include simple leaky-integrate-fire models [50] that models a neuron potential with a RC leaky integrator to more complex Hodgkin-Hexley neuron models [51] that are described through capacitance coupling of each neuron compartments' potentials. These complex, biologically modeled neurons are different from the point-neuron model used in typical DNN models, where real number inputs and real number outputs are used for its inputs and outputs, and its behavior described by simple multiply-and-accumulate (MAC) operations.

In addition to neurons being modeled from real biology, the learning rules could be adapted from biology as well. For instance, spike-timing dependent plasticity (STDP) rules [22] are derived from biological observation of single neuron's synaptic weight changes according to the spike pattern it receives. While the STDP learning algorithm is based on the observations of how a single neuron's synpatic change are altered with exterior stimulation, the composition of neural network hierarchy is another point of consideration for bio-plausibility. For example, the unidirectional nature of synaptic connections between neurons suggests that back-propagation learning rule is not directly plausible inside the biological brain, known as the weight transport problem [23]. Some works [23, 24] solves this problem using separate feedback paths for training, relieving the need for symmetric, bidirectional synaptic connections. These learning rules could also be considered bio-plausible in the sense that they each explains a structural problem in modern deep learning and provide network learning structures that are consistent with biological observations.

In the neuromorphic processor designed in this research, the goal is to retain only the energy efficient properties of neuromorphic learning rules while simplifying computational requirements in the neuron models. For instance, while spiking input-outputs are maintained in the neuron model, simple accumulate operations are chosen for behavior of the neuron rather than using differential equations that describe voltage potentials. The learning rule is modified to take place in a single-shot setting with stochasticity, while maintaining spike-based, local learning rules. The *locality* of the learning rule indicate that error signals do not propagate through the cascaded layers in the learning process: rather, the spiking information from the topmost (output) neuron is fed directly to each of the hidden neurons, resulting in a propagation-free learning. Through these modifications, the goal of obtaining the learning properties that are desirable in hardware implementation is obtained by selecting energy-efficient characteristics in both the biologically inspired algorithms and the modern deep learning. The author speculates that this prevents quantization errors inherent in back-propagation from accumulating while propagating through multiple layers, which provide better tolerance against quantization in the feedback error paths.

4.1.2 Top Level Architecture

Based on the algorithm modifications made on the Segregated Dendrites algorithm discussed in section 2.1, a neuromorphic learning system is designed, fabricated, and verified in silicon level through custom printed circuit boards (PCB) and a host FPGA. A top-level diagram is shown in Figure 4.1. The system has a fixed network architecture, with three layers containing 200-200-10 neurons each. Global data transmission is minimized, using only spikes between the neuron layers in both feedforward and feedback paths. This is realized through the inherent nature of spiking neural networks (SNN) that only communicates with spike (1-bit) information, including the feedback pathways for learning. As shown in Figure 4.1, signals are sent through serialized single-bit spikes between layers, which are implicitly encoded in the clock cycle that the spikes are being sent to be decoded for the spike locations. All neurons implemented in this system were embedded in full digital circuits. While analog neurons were implemented in other works in neuromorphic learning systems [25, 26, 42], analysis on the modified segregated dendrites algorithm implemented in this work showed a requirement of at least 7-bit precision in the internal neuron's dynamics for training without performance degradation as shown in table 4.1, which indicate that the precision analog neurons provide are not enough for robust learning. As the nature of our

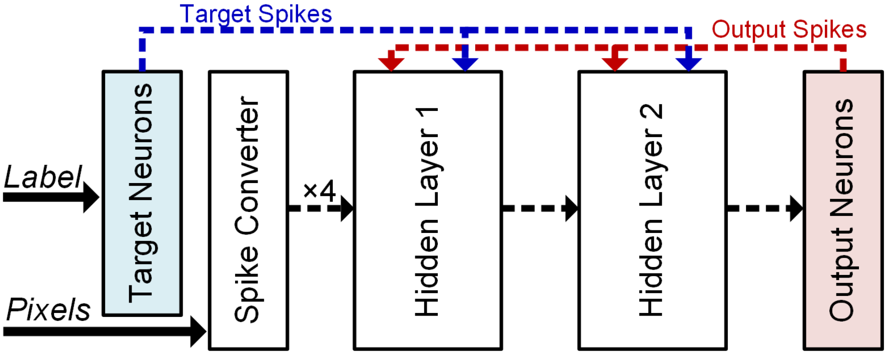


Figure 4.1: A top-level view of implemented neuromorphic system.

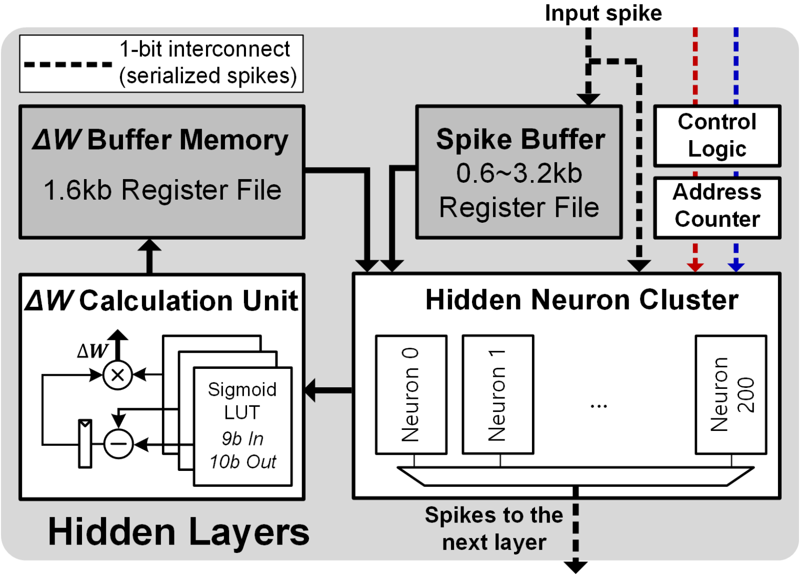
Table 4.1: Performance of SS-MSD algorithm With Different Internal Bit Precision

Internal Bit Precision	**MNIST Test Accuracy**
4 bits	48.80%
5 bits	62.97%
6 bits	96.36%
7 bits	98.04%
8 bits	98.01%

algorithm is highly stochastic, we implemented digital neurons with internal signals represented in 8-bit fixed point arithmetic for higher robustness.

Input pixel data are converted to spikes stochastically using Bernoulli sampling with p equal to pixel density during training, and used a fixed threshold (=0.5) comparing the pixel density and translating to '1' if pixel density is higher than the fixed threshold, and '0' otherwise during inference. The four different layers (input, 2 hiddens, and an output/target layer) are connected serially for transmitting and receiving information of a neuron,for processing both the forward phase (feedforward spikes) and the target phase (feedbackward spikes).

The block diagram of the components in the hidden layer is shown in Figure 4.2. It consists of spike buffers which temporally stores the spiking patterns received from the

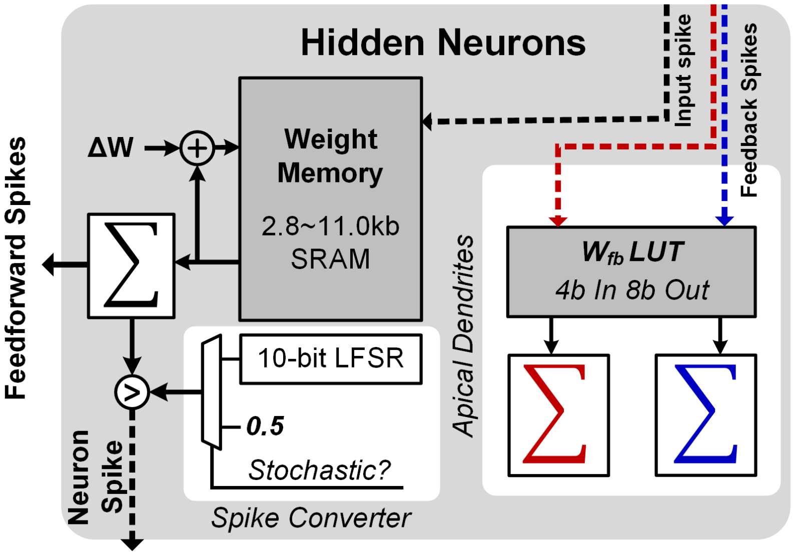


©2020 IEEE

Figure 4.2: Block diagram of components in a hidden layer.

prior layer that is used for finalizing the weight update values, update value calculation unit with sigmoid functionality implemented through 9-bit in, 10-bit out lookup tables, and the hidden neurons. The final weight update values generated by the calculation unit is buffered inside the register files, with updates taking place when the corresponding neuron has spiked for the corresponding image. This hidden layer serially accepts spiking information from the previous layer, interleaved in the cycle timing domain for implicitly converting serial spikes to the corresponding specific neuron in that time cycle. In summary, a hidden layer consists of 200 hidden neurons, selection mux with multiple stages for selecting a spiking neuron, SRAM buffer for holding spiking information, ΔW buffer memory for holding weight update values, and three fixed lookup tables implemented in combinational logic for 9 bit inputs and 10 bit outputs. The bit precision for the lookup tables were chosen from software simulations with emphasis on using the smallest input bit precision that did not show training performance degradation.

The block diagram for the hidden neurons is shown in Figure 4.3. Note that in



©2020 IEEE

Figure 4.3: Block diagram of components in a hidden neuron.

actual implementation, 5 hidden neurons share the physical SRAM implemented as the weight memory for increased area efficiency, as wider SRAMs provide higher area efficiency with shared resources, and the access patterns for the SNN neurons are highly predictable with fixed access patterns across shared neurons. While the sigmoid functions that were used for calcuating the weight update values were prone to numerical errors and had to be implemented in lookup tables with 9-bit input and 10-bit output precision, a hard sigmoid function (approximating sigmoid function) is used in place of using actual sigmoid activations for more efficient processing, described by the equation 4.1 below, where V_s refers somatic dendrite potential, or the accumulated value inside the neuron.

$$Spike\ Probability = clip(0.2 * V_s, 0, 1) \tag{4.1}$$

The spike probability generated with the somatic potential put thorugh the hard sigmoid function is compared against random values generated by 10-bit linear feedback shift registers (LFSR) for generating stochastic spikes during training, while a fixed threshold of 0.5 is used to determine spikes ('1' if probability is higher than the threshold 0.5) during inference. These feedback spikes are transmitted to the next layer, where they are accumulated to the next layer somatic dendrites potentials with their respective feedforward weights decoded by the spike timing. In order to process the feedback pathways, these neurons also hold embedded fixed random weights that are unique in each of the neurons at initialization. These fixed random weights accumulate to their apical dendrites for processing the weight updates.

4.1.3 Lock-Free Weight Updates

The first design technique that is implemented in the neuromorphic processor is using lock-free parameter updates for parallel processing of forward phase, weight update value calculation, and parameter updates. This lock-free updates provide a speedup of ×4 in our pipeline scheme, which would require processing stalls incurred from waiting for weight updates before processing the next images.

As shown in Figure 4.4, during online training of neuromorphic processor the input images are propagated through the hidden dimensions to generate feedback spikes. During this propagation, instead of waiting for finalized updated weight values, next images are processed in a lock-free manner. This is equivalent to processing neural networks with updates being delayed for some extra images are shown: for example, during processing of the nth image, update values for $(n-3)^{th}$ image are being calculated, with the update for $(n-4)^{th}$ image actually takes place. In other words, update for nth is only takes effect after $(n+4)^{th}$ image.

While this parallel processing reduces the clock cycle overhead for training drastically by ×4 in our implementation, it needs to be verified that this processing style does not impact training accuracy, as it changes update behaviors. In our simulations, the lock-free implementation does not degrade MNIST classification accuracy, as shown on the bottom left corner of Figure 4.4. This is in line with the observations made in prior works [36, 37] that address implementing stochastic gradient descent (SGD)

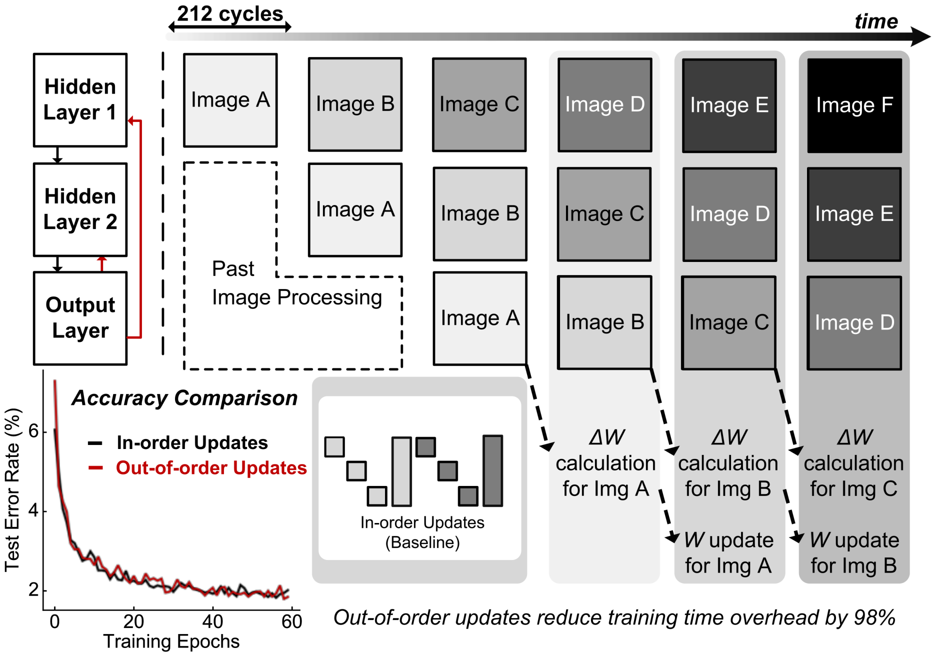


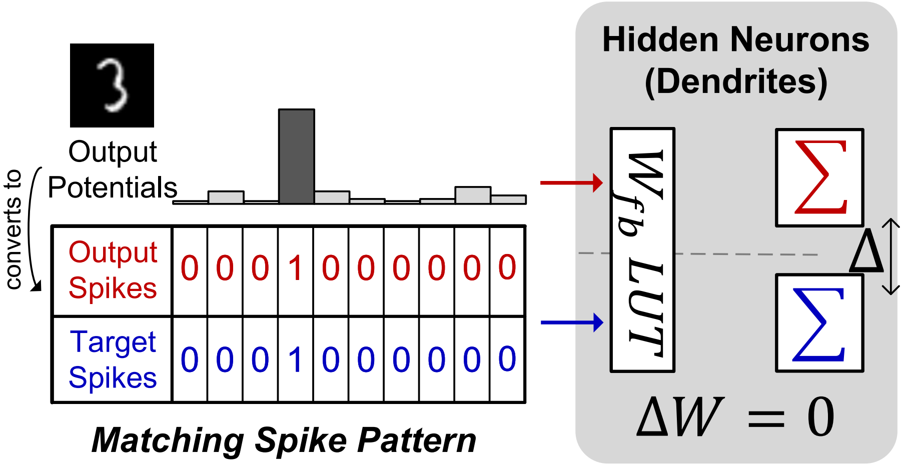
Figure 4.4: Diagram of lock-free parallel processing.

algorithms in distributed systems: delaying parameter updates within few training examples do not impact overall training performance.

Another benefit of this parallel processing is that it could reduce the total number of reads required for parameter updates, as our processing pipeline makes use of the feedforward weight read out for processing the forward phase and makes updates on that specific synaptic weight. Therefore, $w + \Delta W$ takes place when the corresponding weight is read out for forward processing, reducing 1 read operation per synaptic weight that needs update.

4.1.4 Update Skipping Mechanism

In the case of SGD updates with back propagation, weight update values become smaller in absolute size as training progresses so that model output becomes close



©2020 IEEE

Figure 4.5: Illustration of a case where output and target spikes match exactly, where update values become zero.

to the targeted output. In our modified segregated dendrites algorithm, this observation becomes more extreme, leading to *sparse* weight updates where up to 90% of the total update values for a given training epoch are equal to zeros. This sparsity is an inherent property in the update algorithm that is chosen for the modified Segregated Dendrite algorithm. From equation 2.11, when the apical dendrite potentials during forward and target phases (V_{a_f} and V_{a_t}) match exactly, update values are all equal to zero. This case is not rare if training progresses, as the apical dendrites are derived from target and output spikes, which are only 10 bits of information each. Thus, as illustrated in Figure 4.5, when these target and output spikes match exactly (e.g. the neural network responds correctly with only the output neuron that correspond to the label '3' firing, and all other neurons not firing to match the one-hot encoded target spikes), we can skip update values altogether without altering the behavior of the modified segregated dendrites algorithm. Therefore, we could speculate sparsity in weight updates *if* the network successfully learns to classify images to their correct labels exactly. Notice that matching spike patterns are not equivalent to a correct classification: multiple neurons could still fire for a correctly guessed image, when multiple neurons have been

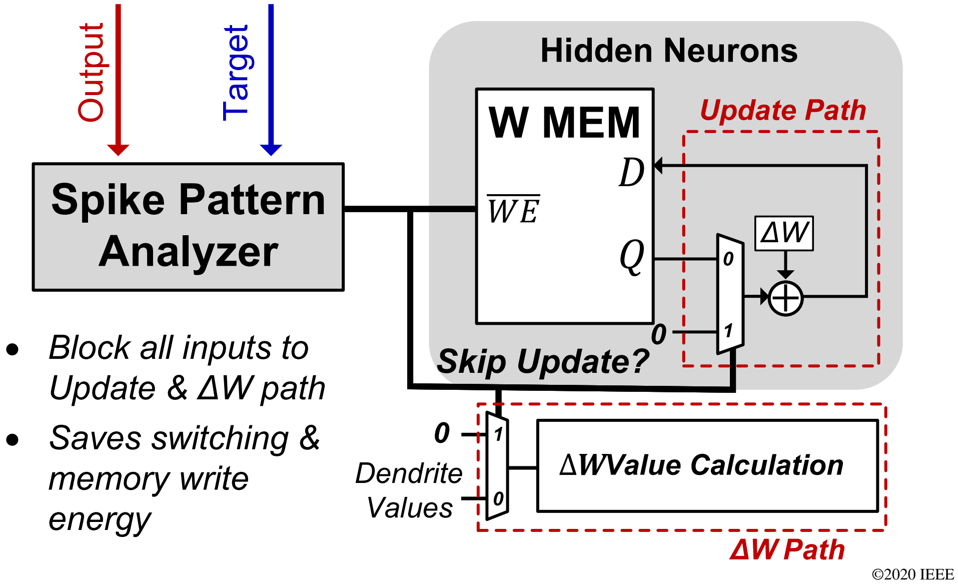


Figure 4.6: Disabling the update path through monitoring the spike pattern.

above the firing threshold but the output neuron corresponding to the correct label was the one with highest potential. Therefore, it is important for the neural network to *discourage* wrong labels from spiking as well for sparse update weights, and the update sparsity rate is upper bounded by the classification accuracy (as it holds true that a matching spike pattern always classifies images correctly, but the converse does not hold true).

This observation is exploited through logically gating the update value calculation path and disabling SRAM write paths on the weight update pipeline when spike match conditions are met, as illustrated in Figure 4.6. Energy consumption in the system benefits from logically gated update value calculation unit with long (¿200) clock cycles where combinational logic gates are not switching. SRAM writes being disabled also benefits power consumption with reduced write operations that overwrites contending SRAM cells. In our software simulations, as training progresses, update skip rate starts at around 3.8% and increases to 81.7% after training on 60000 images. We could expect a power reduction in training progression: these results are measured and analyzed

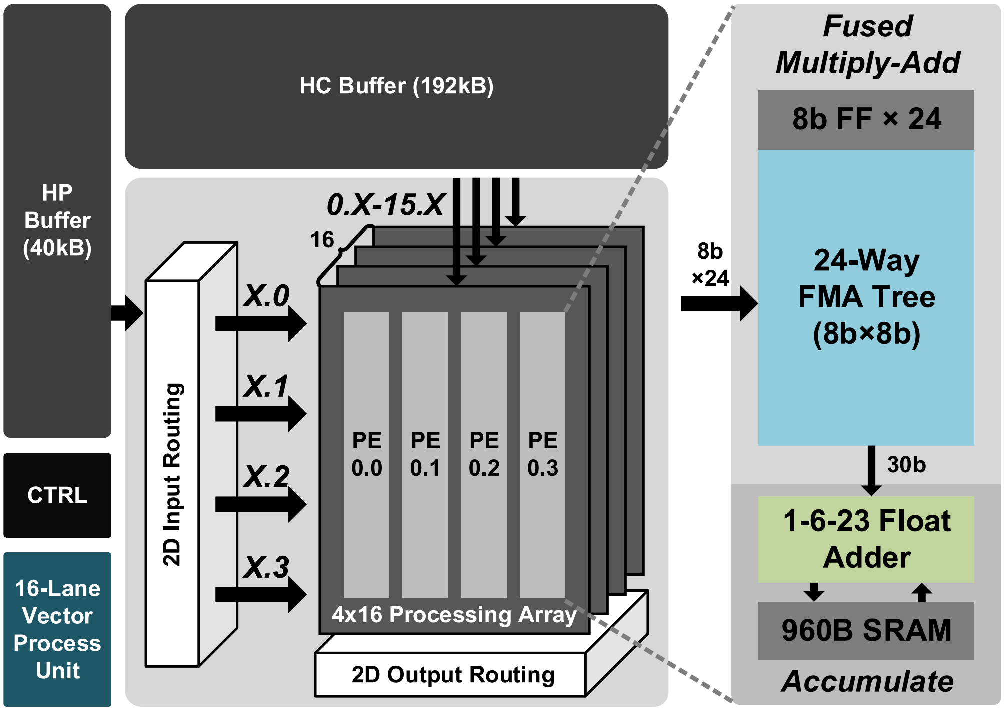
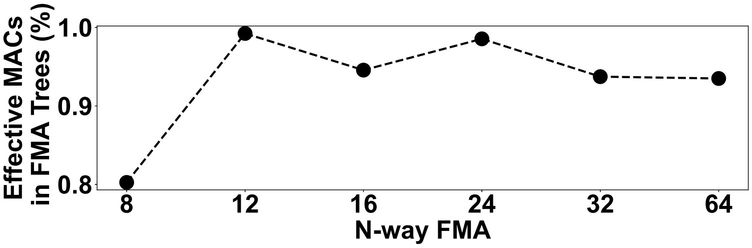


Figure 4.7: Top level block diagram of the DNN training processor.

in more detail in section 5.1.1.

4.2 Low-Precision DNN Training System

In this section, the overall architecture of the system and the design techniques that are implemented on a fabricated integrated circuits chip for DNN training processor that implements the floating point arithmetic introduced in section 3 is described in more detail. The top-level architecture choice, as well as the design functions and novelties such as flexible spatial routing on efficient tree-based processing architecture, custom acceleration instructions, and bank composition are introduced in this section.



©2021 IEEE

Figure 4.8: Ratio of valid MACs with varying N in N-way FMA trees in ResNet-18 processing.

4.2.1 Top Level Architecture

Based on the newly proposed 8-bit floating point numeric system that were discussed in chapter 3, an end-to-end custom DNN training accelerator is designed in a digital integrated circuits system and verified for real time training on a silicon level.

The top level block diagram of the DNN training processor is shown in Figure 4.7. The main processing for DNNs occur on a tree-like processing structure described in section 3.4. This tree performs 24-way Fused Multiply Add (FMA) operation on two 8-bit vectors with length of 24, reducing down the FMA result to a custom FP30 (1.6.23) format. This processor contains 64 of these FMA trees organized in a 4×16 structure, which is equivalent to 1,536 multiply-accumulate (MAC) operations per cycle. The reasoning behind using a 24-way FMA tree rather than using some power-of-2 (such as 32 or 64-way FMA) is to ensure maximum utilization of the trees in our processor for optimizing to varying convolution window sizes, as 24 is a multiple of 1, 2, 3, and 4 (which are all kernel sizes used commonly accounting for stride of 2). Figure 4.8 shows the utilization rate of the FMA trees for processing ResNet-18 with processing architectures with varying N in N-way FMA. While 12-way shows slightly higher utilization compared to 24-way FMA trees, we adopt 24-way as higher N results in better energy and area efficiency.

For spatial processing of convolutional layers in both forward and backward phase, routing units are placed on input and output vectors of the tree groups, reducing on-chip memory access. A scratchpad for holding accumulation values, as well as being served as vector memory for the vector processor is implemented with 30-bit wide 256-word deep dual port register file, with each scratchpad corresponding to an FMA tree, totaling 64 scratchpads with 60kB of memory. Additionally, two buffer units, high-performance (HP) bank with 40kB of memory and high-capacity (HC) bank with 192kB of memory support caching of activations and weights to reduce external memory read and writes. Combined with the 1kB FIFO that handles direct memory access (DMA) transactions, our chip contains a total of 293kB in terms of on-chip memory. Lastly, a 16-lane floating point vector processing architecture shown in Figure 4.9 support auxiliary operations required for end-to-end DNN training, such as pooling operations, weight updates, etc. In more detail, a fetched instruction will be decoded to determine which executor (FP vector execution unit, integer execution unit, matrix multiplication and convolution execution unit) will process on the instruction. For example, FP vector execution unit will perform A*X + Y operations (AXPY), while matmul and convolution execution unit uses dedicated finite state machines (FSMs) dedicated for controlling the FMA tree units, routers, and buffers to process instructions according to their hyperparameters. Moreover, these FSMs are supported with additional memory address generators with uops issued by the control logic in the FSMs. The uops are a selection from the group of address generation patterns, such as partial tiled weight loading the HC buffers, prefetching grouped row loading to the HP buffers, etc. Given the uop type, target memory section, and base addresses, these memory generators create the memory address patterns and handles memory transactions through direct memory access. The direct memory access is implemented with FIFOs and a wide I/O that connects to external FPGA design with DDR3-SDRAM.

The FP vector unit is consisted of 16 reconfigurable vector registers of 128 bits each (for re-configuring into 16×8b or 8×16b vectors), a variable-precision AXPY unit, full-precision FPU for supporting IEEE-compliant operations on scalar FP32 data, data

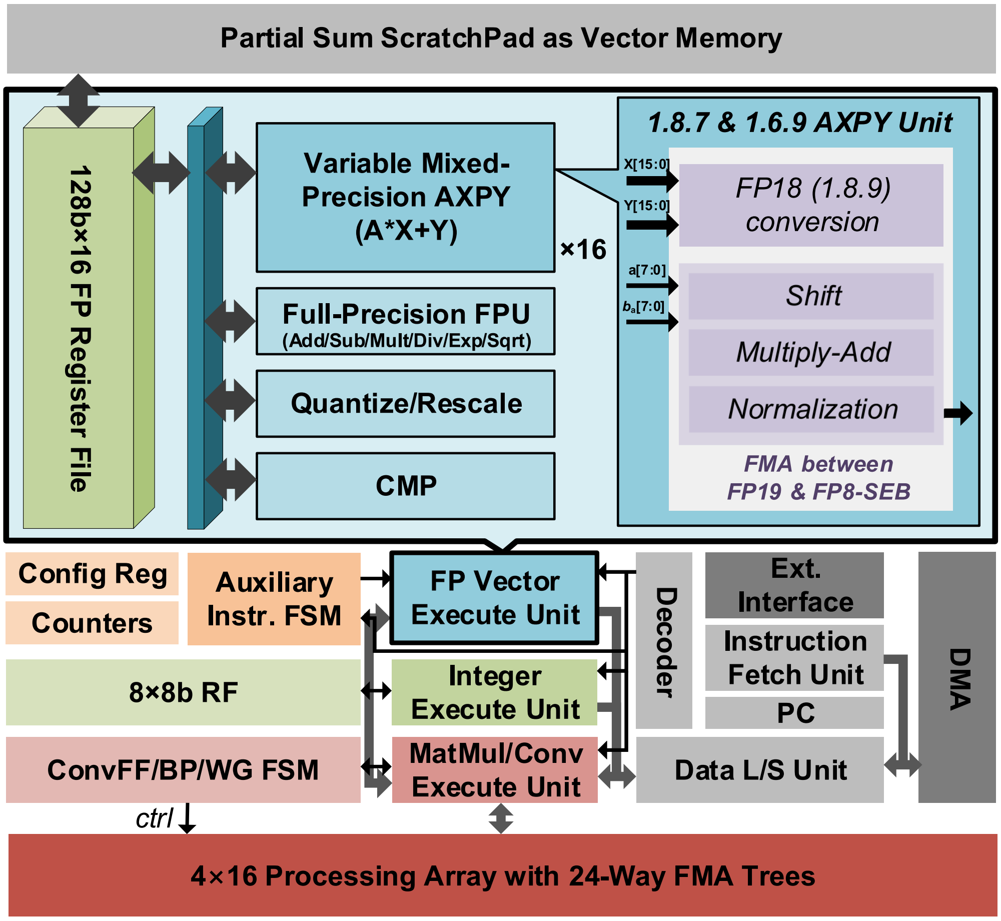


Figure 4.9: Block level view of the 16-lane vector processor.

format casters that changes the data formats to or from any of the 5 data types used in this processor (1.4.3 FP8-SEB, 1.8.7 brain float, 1.6.9 FP16, 1.6.23 FP30 accumulator, and 1.8.23 full-precision), and a comparator. The AXPY unit provides variable precision between the two 16-bit floating point precision used in this processor through implementation as an FP18 AXPY unit and including format converters. In more detail, the AXPY unit could accept up to three 16-element vectors with (X, Y) in FP16 format, and a in FP8 format. Additionally, a scalar b is accepted to account for scale. 8 out of the 16 AXPY units adds support for stochastic rounding [15] with LFSR units. Note that while actual hardware could only perform on (X:FP16, Y:FP16, a:FP8) format, the microarchitecture in this processor accepts arbitrary precision out of the 5 data formats as the FP vector execution pipeline implicitly puts the vector registers to the data format casters before sending the data to the AXPY unit. The dedicated control units tile the order of computation in convolution with optimized logic, which is described in more detail in section 4.2.4. Moreover, the data load and store unit includes dedicated address generators that could work with memory prefetchers detailed in section 4.2.3 for the tiled computation of convolution, as well as on-chip data transpose unit to facilitate computation during backward and weight gradient phase of DNN training without physically re-loading and storing transposed memory to external DRAMs. While dedicated memory controllers could be implemented for managing external DRAMs, this design currently hosts simple wide 128-bit I/O for direct memory access through external FPGA-based DDR controllers.

4.2.2 Optimized Auxiliary Instructions in the Vector Processing Unit

The vector processing unit, in addition to general-purpose AXPY and element-wise vector operations, supports custom accelerated instructions that handles some of the widely used operations used in training DNNs such as stochastic gradient descents, softmax, and pooling. This custom accelerated instructions utilize bypass routing and additional control logic that avoids data hazards and could make use of multiple functional units in a single pipeline stage to maximize processor utilization and minimize external data access.

The stochastic gradient descent (SGD) algorithm is often used for optimizing state-of-art DNNs with given weight gradients. This algorithm is often aided by weight decay and momentum. In our notation, SGD algorithm update takes weight gradient (gW), current weight (W_t), current momentum (M_t), learning rate (lr), weight decay constant (d), and momentum constant (μ) as input to produce updated weight (W_{t+1}) and updated momentum (M_{t+1}). Equation 4.2 denotes the effect of weight decay, while equation 4.3 calculates the updated momentum, which will then be used in equation 4.4 to generate the final updated weights.

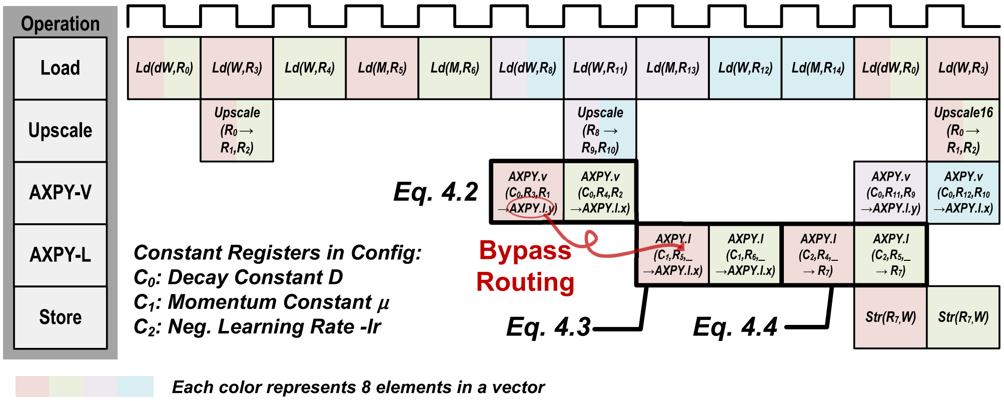
$$gW' = d * W_t + gW \tag{4.2}$$

$$M_{t+1} = \mu * M_t + gW' \tag{4.3}$$

$$W_{t+1} = W_t - lr * M_{t+1} \tag{4.4}$$

Note that in our training scheme on the FP8-SEB processor, we use a separate 16-bit copy of trained weights and 16-bit momentum, similar to prior literature [31].

In a naïve SGD implementation, one would require 1 upscaling operation for casting 8-bit weight gradient (gW) to 16 bits for casting to AXPY operations, and additional 3 AXPY operations. Moreover, stall cycles are required to wait for intermediary results to be written to FP registers. In our optimized SGD implementation, routing paths are created such that: (1) Upscaling unit and AXPY unit is activated at the same time, (2) Special routing allows AXPY results to be bypassed directly to AXPY inputs for removing stalls for data hazards. The implementation paths that includes the fast-forwarded paths are shown in Figure 4.10. In more detail, the AXPY units are grouped to two different types: (1) AXPY-v, where the v stands for *vanilla*, and (2) AXPY-l, where the l stands for *LFSR*, which also includes an LFSR unit to support stochastic rounding. Moreover, the upscaling unit that casts 1.4.3 FP8 elements to either one of the FP16 formats (1.8.7 or 1.6.9) could be utilized within the same cycle. Through these optimizations, our specialized SGD instruction require 31.7% less clock cycles to complete compared against SGD implementation through combinations of AXPY and upscaling instructions. Moreover, to better support average pools and var-mean calculation used for batch normalization, we allow vector processor to access the FP30 accumulators used in the main DNN processing logic for high-throughput computation. This is realized through the custom channel-wise accumulation and squared accumulation instruction.


Operation
Load
Upscale
AXPY-V
AXPY-L
Store
Ld(dW,R_0)
Ld(W,R_3)
Ld(W,R_4)
Ld(M,R_5)
Ld(M,R_6)
Ld(dW,R_8)
Ld(W,R_{11})
Ld(M,R_{13})
Ld(W,R_{12})
Ld(M,R_{14})
Ld(dW,R_0)
Ld(W,R_3)
Upscale ($R_0 \rightarrow R_1,R_2$)
Upscale ($R_8 \rightarrow R_9,R_{10}$)
Upscale16 ($R_0 \rightarrow R_1,R_2$)
Eq. 4.2
AXPY.v ($C_0,R_3,R_1 \rightarrow$AXPY.l.y)
AXPY.v ($C_0,R_4,R_2 \rightarrow$AXPY.l.x)
AXPY.v ($C_0,R_{11},R_9 \rightarrow$AXPY.l.y)
AXPY.v ($C_0,R_{12},R_{10} \rightarrow$AXPY.l.x)
Bypass Routing
AXPY.l (C_1,R_5,_ $\rightarrow$AXPY.l.x)
AXPY.l (C_1,R_6,_ $\rightarrow$AXPY.l.x)
AXPY.l (C_2,R_4,_ $\rightarrow R_7$)
AXPY.l (C_2,R_5,_ $\rightarrow R_7$)
Str(R_7,W)
Str(R_7,W)
Constant Registers in Config:
C_0: Decay Constant D
C_1: Momentum Constant μ
C_2: Neg. Learning Rate -lr
Eq. 4.3
Eq. 4.4
Each color represents 8 elements in a vector

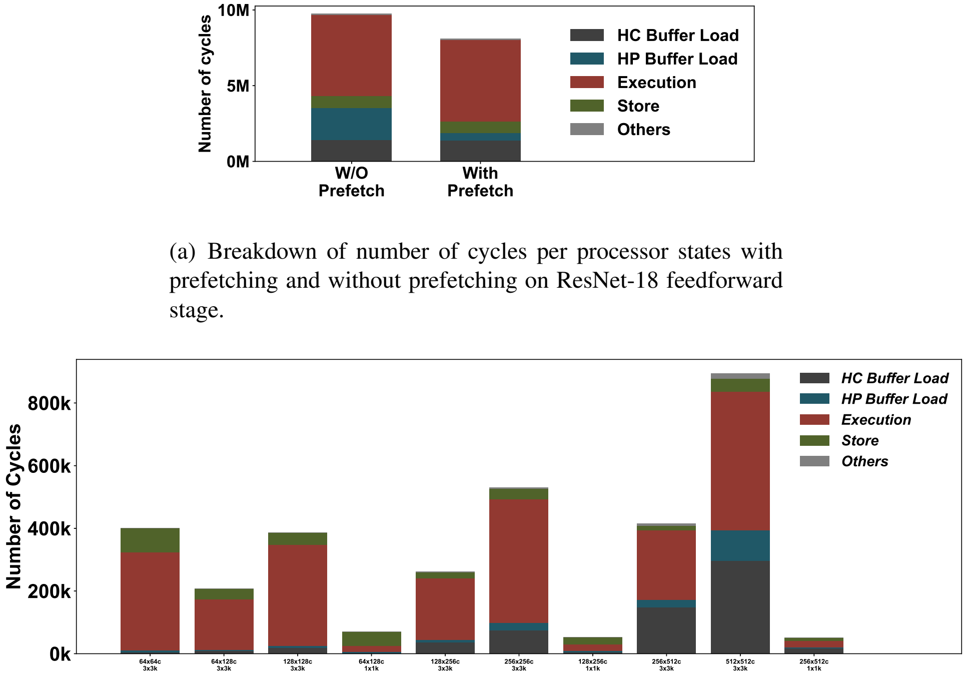
©2021 IEEE

Figure 4.10: Cycle-by-cycle operation of the optimized SGD instruction.

4.2.3 Buffer Organization

The on-chip memory on the DNN training processor is categorized as high-performance (HP) buffer with 40 kB of memory, high-capacitance (HC) buffer with 192 kB of memory, and local scratchpad with 60 kB of total memory. The HP buffer contains 32 two-port SRAMs with 5B word and 256 entries. This buffer is usually used for caching activations and gradients and supports memory pre-fetching feature for reducing memory stalls. This prefetching is enabled through using 4B of the 5B word as 'active' region and the remaining 1B as 'pre-fetching' region. Write word masks allow writes to only the pre-fetching region when this feature is enabled. Out of the 5B readout words, the active 4B is selected through barrel shifters.

Through allowing explicit prefetching in activations and gradients placed in HP buffer, memory stalls are reduced for memory operations from external memory to HP buffer. The effect of this prefetching is analyzed in Figure 4.11(a), where latency for feedforward stage of ResNet-18 model on ImageNet sample data with batch size of 4 is broken down to the processor's operation types. In the left side of the graph, prefetching is disabled. On the right side, prefetching is enabled, allocating HP buffer load time (shown in blue) to active processing time (shown in red). Memory latency is reduced by ×1.63, whereas the overall latency is reduced by ×1.20. The breakdown of proces-



(a) Breakdown of number of cycles per processor states with prefetching and without prefetching on ResNet-18 feedforward stage.

(b) Profile results per layer in ResNet-18 with prefetching enabled

Figure 4.11: Breakdown of processor states in terms of number of cycles.

sor states per layer is shown in Figure 4.11(b), showing the varying processor states by different layer configurations in the ResNet-18 model. Noticeably, in the lower layers where the memory access counts are dominated by activations, the prefetcher implemented for the HP buffer mitigates the memory accesses successfully. While prefetching for HP buffers occur with writing on the masked *row* of the SRAM data such that all the *columns* of the SRAM is utilized, requiring elaborate masking schemes for avoiding data collision, more simple methods could be utilized to reduce memory stalls by segmenting buffer sections into active *columns* and deactivated columns that are currently being used for prefetching. This method is implemented for the scratchpads that holds the final accumulated values. For example, columns 0 through 127 out of the 256 columns available are marked for pushing final accumulated value to the external memory. In this case, instead of waiting for the write pushes to finish, the processor

Table 4.2: Capacity and Bandwidth of Banks*

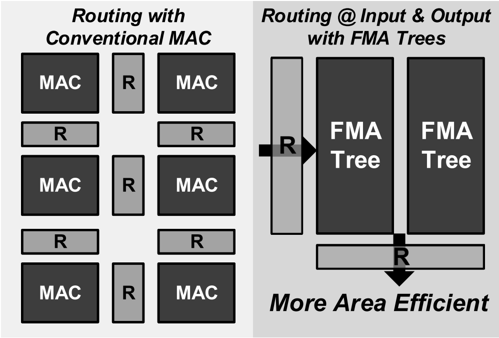
	HC buffer	**HP Buffer**	**ScratchPad**
Configuration	16×(24-bit×512)	24×(40-bit×256)	64×(30-bit×256)
Capacity	192kB	40kB	60kB
Max Bandwidth*	2.88GB/s	4.32GB/s(@write) 17.28GB/s(@read)	43.20GB/s
Prefetching	No	Yes	Yes (Store)

*At 180MHz clock frequency.

proceeds to the next stage of computations while only using columns 128 to 255, as the first 128 columns are not available until the memory interface logic has finished the qued task of storing the scratchpad data to the external memory. Write stalls only occur in the case when the computation stage requires the scratchpad column indexes 0 to 127 before write push is finished. While similar methods could be implemented for loading stage of the HC buffer, it incurs an overhead where HC buffer could not be utilized to full extent, resulting in smaller tiles of computation and larger external memory accesses. Thus, we do not implement such prefetchers for the HC buffeer. As a result, in the upper layers where the memory accesses are dominated by the model parameters, the HC buffer load stages could be seen to take up noticeable amount of processor cycles as no prefetcher is implemented on the HC buffer to mitigate memory stalls.

In the three different categories of on-chip memory, data are organized in a manner such that data re-use are maximized so that data movements could be reduced. For instance, during convolutional feedforward operation, HC buffers caches weight data loaded from external memory. In order to reduce HC buffer access, weights that are read out are shared across 4 processing units (single row) in the 4×16 processing array.

Activations are cached in the HP buffer and are processed spatially so that data read/writes are reduced by the size of the kernel (kernel height × kernel width) in convolution layer processing compared to naïve implementation that does not makes use


Routing with Conventional MAC
MAC
R
MAC
R
R
MAC
R
MAC
R
R
MAC
R
MAC
Routing @ Input & Output with FMA Trees
R
FMA Tree
FMA Tree
R
More Area Efficient

©2021 IEEE

Figure 4.12: The routing at inputs and outputs for tree based processing, compared against conventional routing units that is implemented between processing elements.

of data spatiality. This spatial processing is enabled through the input-output routing units that is discussed in the next section and will be analyzed in comparison against another work that utilize spatiality in systolic processing array for evaluation of the routing method.

4.2.4 Input-Output 2D Spatial Routing for FMA Trees

In input-output spatial routing scheme that is proposed in this research, routing units are restricted to the inputs and the outputs of the 4×16 24-way FMA tree processing array. In the inputs side, 4 'rows' of 24-element vector inputs are routed through barrel shifters, muxes and shift registers to allow spatial routing in convolution layers. Similarly, the 4×16 outputs from the processing array are routed for spatial processing, especially for transposed convolution and convolutional backward processing. The conceptual diagram for this routing style is contrasted against intermediary routing style in conventional MAC-based systolic architectures in Figure 4.12.

This routing style allows routing with convolutional spatiality taken into account for optimized data movements and reduced buffer accesses in tree based processing architectures for DNN training. To the best of our knowledge, while other work [38] has also proposed using energy-efficient tree based processing for DNN workloads, this work is the first to present routing mechanisms that allow convolutional spatiality in tree based processing architectures in both convolution feedforward and convolution feedbackward stages.

For example, during convolutional feedforward stage, the input routers allow reduced data access requirement on the HP buffers through the input routing units. The 32 SRAMs in the HP buffers are given group ids, with the number of channel groups given by equation 4.5, the number of SRAMs per channel group given by equation 4.6, the group id of a specific channel c calculated through equation 4.7, and the SRAM id of that channel given by equation 4.8.

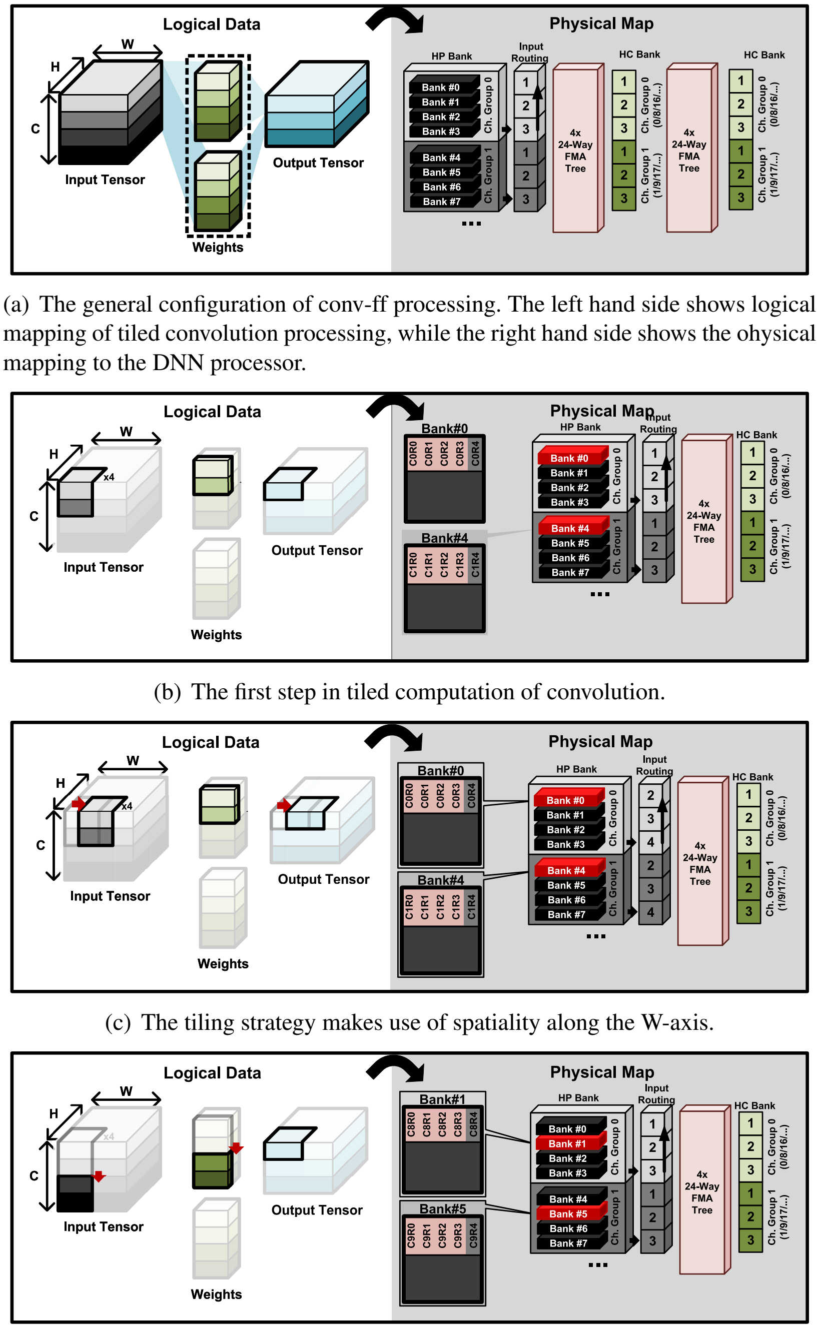
$$N_{ChannelGroup} = floor(24/KernelWidth) \tag{4.5}$$

$$N_{BankPerGroup} = floor(32/N_{ChannelGroup}) \tag{4.6}$$

$$ID_{ChGroup}(c) \equiv c \mod N_{ChannelGroup} \tag{4.7}$$

$$ID_{SRAM}(c) \equiv (c \mod N_ChannelGroup) \mod N_{BankPerGroup} \tag{4.8}$$

The case for convolutional feedforward processing order when the convolution kernel sizes are 3×3 is shown in Figure 4.13. The input activation tensor with (Batch, Channel, Height, Width) shape is shown on the left side of Figure 26 (a). This input tensor is split along the channel axis and mapped to specific SRAMs according to equation 4.8. Since the kernel width is 3, there exist 8 channel groups following equation 4.5. One channel in each of the channel groups are selected to be mapped to the processing array, so that 8 channels are processed simultaneously as seen in Figure 4.13(a). The SRAMs in HP buffer contain 4 active bytes in a word, which we call 4 processing rows. The input activation's 'rows' (image patches along the height axis) are mapped to these processing rows as denoted in the right figure of Figure 4.13(b). Effectively, 4 rows along the height axis in an input activation map is processed at a time. In the next cycle, convolution window slides along the 'width' axis, as seen in Figure 4.13(c). After processing all of the input activations in the 'width' axis, the next



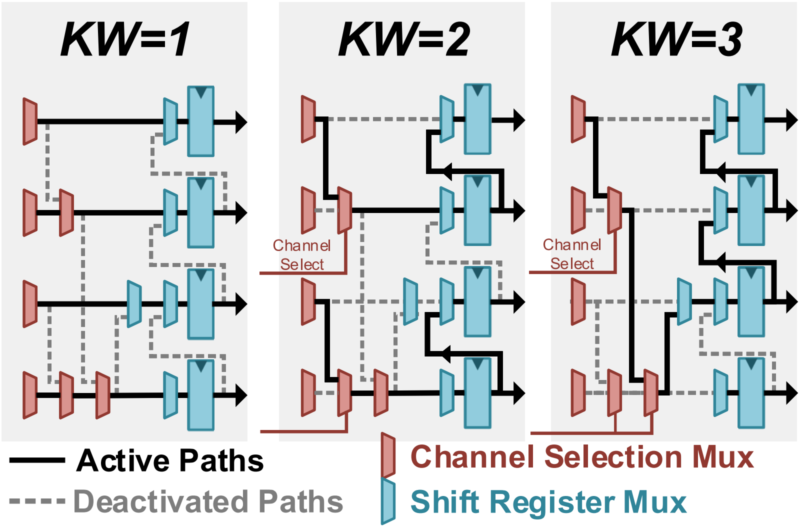
(a) The general configuration of conv-ff processing. The left hand side shows logical mapping of tiled convolution processing, while the right hand side shows the ohysical mapping to the DNN processor.

(b) The first step in tiled computation of convolution.

(c) The tiling strategy makes use of spatiality along the W-axis.

(d) After finishing a row group, the next rows are sequentially processed.

Figure 4.13: Illustration of convolution feedforward processing steps in 3×3 kernel size.



©2021 IEEE

Figure 4.14: Input routing unit in implementation for different cases of kernel widths for illustration.

SRAMs in the processing group is selected until all of the input channels have been accumulated, as seen in Figure 4.13(d).

In summary, the input router performs the following operations to enable this processing order: (1) it first aligns the four active 'rows' along the height axis of the input activations from the HP bank through barrel shifters, then (2) it selects the active SRAMs from the active input channels being accumulated through 3-stage muxes, and finally (3) shifts the selected rows and channels through the width axis considering the sliding window of convolution. The diagram of the implemented input router is shown in Figure 4.14, showing input routers in different configurations of kernel widths.

With the implemented input router, the complete pseudo code for this computational ordering in convolutional feedforward instruction is shown in Algorithm 1. As HC buffers may not be large enough to accommodate all of the weights in a convolution parameter, we first split the output channels with the number of maximum output channels, as in line 1 of Algorithm 1, iterating for $N_{OCSplit}$ times. During the iteration, in order to ensure that no excessive memory loads and stores occur, all of the output ac-

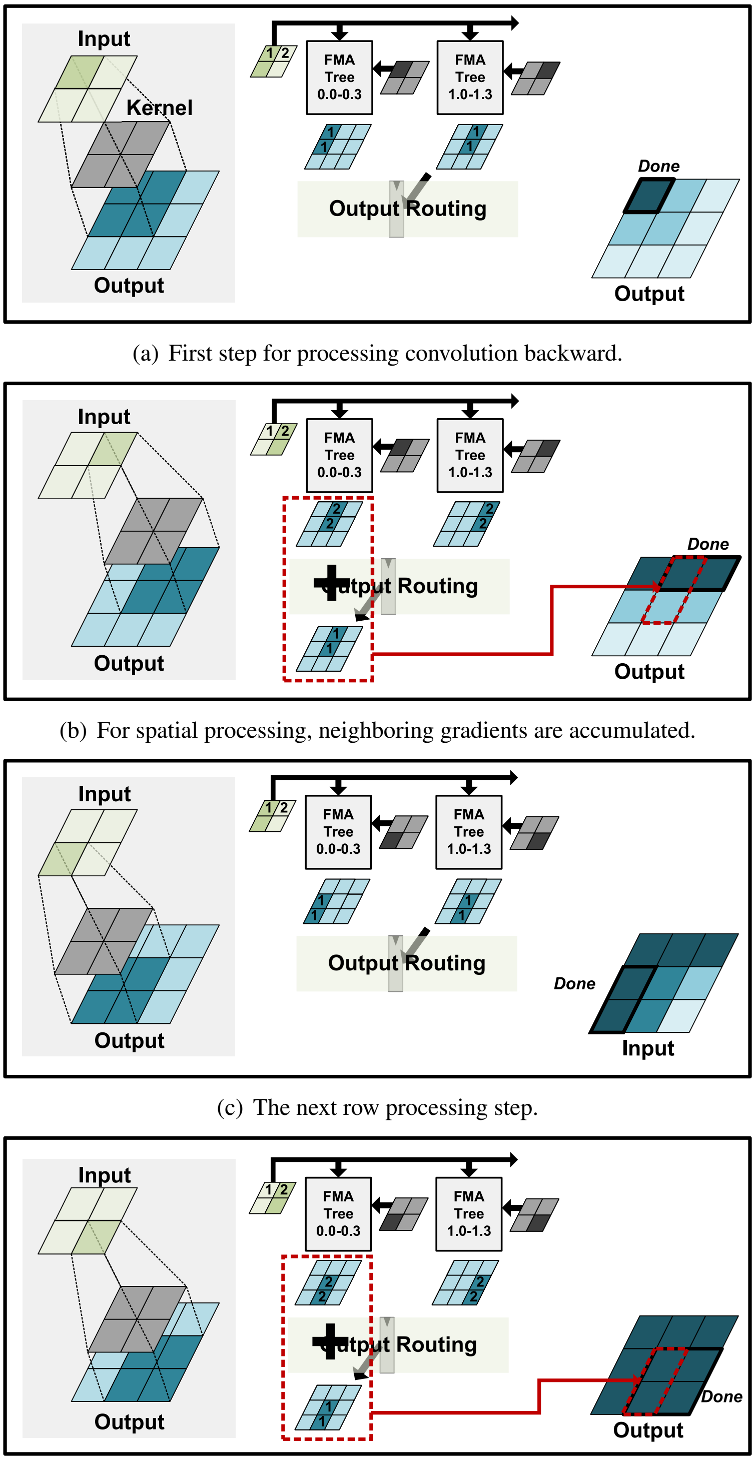
Algorithm 1 Conv-FF Implementation

```
1:  $N_{OCSplit} = Ceil(\frac{WeightSize}{HCBufferSize})$
2:  Initialize $O[Batch, O_{Ch}, O_H, O_W] \leftarrow 0$
3:  for $O_{chSplit} \leftarrow 1$ to $N_{OCSplit}$ do
4:      Load split output channel weights to HC Buffer
5:      $StartO_{ch} = \frac{O_{ch}}{N_{OCSplit}} * (O_{chSplit} - 1)$
6:      $EndO_{ch} = \frac{O_{ch}}{N_{OCSplit}} * O_{chSplit}$
7:      for $b \leftarrow 1$ to $Batch$ do
8:          for $r \leftarrow 1$ to $InputHeight$ do
9:              for $o_{ch} \leftarrow StartO_{Ch}$ to $EndO_{Ch}$ do
10:                 for $kh \leftarrow 1$ to $KernelHeight$ do
11:                     for $IC_{Split} \leftarrow 1$ to $N_{bankPerGroup}$ do
12:                         Change channel selection mux
13:                         Read Weights to FMA Tree
14:                         $i_{Ch} = N_{chGroup} * IC_{Split}$
15:                         for $p \leftarrow 1$ to $InputWidth$ do
16:                             $O[b, o_{ch}, row, p]$ +=
                                $\sum I[b, i_{ch} : +N_{chGroup}, r, p : +KW]$
                                $*W[o_{ch}, i_{ch} : +N_{chGroup}, kh, :]$
17:                         end for
18:                     end for
19:                 end for
20:             end for
21:             Store Output Row to Ext. Memory
22:         end for
23:     end for
24: end for
```

tivations in output channels of $StartO_{ch}$ to $EndO_{ch}$ are processed completely without requiring offloading high precision intermediary results to the external memory. After weights are loaded to the HC buffer, the processing loop order is as follows: (1) Pixels along the four active rows are processed along the input activation width axis (lines 15-17), (2) the channel selection in the input channel group is changed, while loading corresponding weight with processing input channel, output channel, and kernel height to the FMA trees (lines 11-13), (3) change the processing kernel height (line 10), (4) change the processing output channels (line 9) and store the finished output channel to external memory (line 21), (5) change the active rows along the height axis of the input activation tensor (line 8), (6) change the current processing batch (line 7). The global input routing unit, which consists of barrel shifters, 3-stage channel selection mux, and shift registers, aligns data according to its rows, channels, and spatial positions to allow this type of processing order.

In a similar manner, our processor could implement the feedbackward step of convolution layers (which are equivalent to transposed-convolution) necessary for CNN training through utilizing the output data point routing. As shown in Figure 4.15, feedbackward step of convolution layers (conv-bp) could be considered as a transpose of convolution feedforward and exhibits spatial correlation between the output data of conv-bp operations. There exist spatiality between the computations required for neighboring pixels in the *output* of the conv-bp operation as they share the same pixel location of the input data to complete their computation, as shown on the left-hand side of the illustrations. Therefore, a time-delayed accumulation of the same pixel location results in reducing data read and writes required for conv-bp processing.

In Figure 4.15(a), the accumulated value that is processed by the second FMA tree is delayed by a cycle to be accumulated to the pixel values that are processed as in Figure 4.15(b). The end result is that neighboring accumulation values are added together through the readily available adder without requiring multiple read-outs of input or output data. Similarly, to account for the spatiality along the H-axis in convolution feedbackward, the rows are selected through barrel shifters to be accumulated to the neighboring gradient rows that were computed in the previous steps. For example, in Figure 4.15(c) and 4.15(d), the gradients are being accumulated to the gradient values that were computed in the prior steps in Figure 4.15(a) and 4.15(b). Figure 4.16 illustrates the routing connections required for allowing conv-bp operations to be processed with spatiality without requiring additional adders. The thick black line denotes activated paths, while the dotted gray line denotes the deactivated paths. During convolution feedbackward stage, the output routing unit connects the neighboring



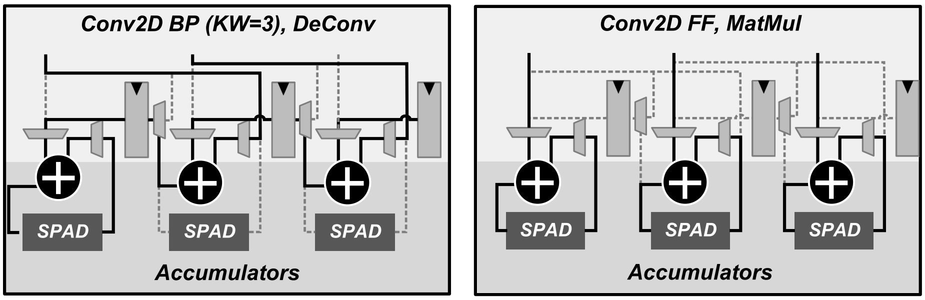
(a) First step for processing convolution backward.

(b) For spatial processing, neighboring gradients are accumulated.

(c) The next row processing step.

(d) For spatial processing, the neighboring rows are accumulated to results from (a) and (b).

Figure 4.15: Illustration of convolution backward functionality.



(a) Output routing unit configuration in Conv-BP, kernel width=3

(b) Output routing unit configuration during feedforward operation

©2021 IEEE

Figure 4.16: Illustration of the 2D output routing unit.

pixels through time-delayed accumulation of gradients with shift registers with length that is equal to the kernel width, variable lengths supported through selection muxes. Moreover, the barrel shifters account for spatiality along the H-axis, accumulating to neighboring rows equal to the kernel height. This case is illustrated in Figure 4.16(a) for a kernel width of 3. By delaying addition of the same input pixel gradient by 1 or 2 cycles, spatiality is effectively achieved as the end result is accumulation of neighboring pixel gradients. While the additional accumulation may seem to incur additional adders for supporting conv-bp operation with spatiality, this could supported without requiring additional adders, as they are performed by sharing the same adders that are used for accumulation during feedforward and matrix multiplication operations. None of the addition operations are mapped to overlapping adders as they could be avoided through simple muxed selection, as seen in Figure 4.16(b).

The pseudo code for the computational ordering in the conv-bp operations are shown in Algorithm 2. Note that the annotation 'I_{grad}' refers to the actual output of the conv-bp function, and 'O_{grad}' refers to the actual input of the conv-bp function, as conv-bp is a transpose of conv-ff. Following the same logic from conv-ff, input channels are split by the maximum number of channels that the HC buffer could accommodate, as seen in line 2, and has a similar processing order to conv-ff implementation: (1) Pixels along the four active rows are processed along the output gradient width

Algorithm 2 Conv-BP Implementation

```
1:  $N_{O_{Ch}} = 16//KernelWidth$
2:  $N_{ICSplit} = Ceil(\frac{WeightSize}{HCBufferSize})$
3:  Initialize $I_{grad}[Batch, O_{Ch}, O_H, O_W] \leftarrow 0$
4:  for $I_{chSplit} \leftarrow 1$ to $N_{ICSplit}$ do
5:      $StartI_{ch} = \frac{I_{ch}}{N_{ICSplit}} * (I_{chSplit} - 1)$
6:      $EndI_{ch} = \frac{I_{ch}}{N_{ICSplit}} * I_{chSplit}$
7:      Load split input channel weights to HC Buffer
8:      for $b \leftarrow 1$ to $Batch$ do
9:          for $r \leftarrow 1$ to $InputHeight$ do
10:             Load Row to HP Buffer
11:             for $i_{ch} \leftarrow StartI_{Ch}$ to $EndI_{Ch}$ do
12:                 for $kh \leftarrow 1$ to $KernelHeight$ do
13:                     for $OC_{Split} \leftarrow 1$ to $N_{bankPerGroup}$ do
14:                         Change BankSelect
15:                         Read Weights to FMA Tree
16:                         $o_{Ch} = N_{O_{ch}} * OC_{Split}$
17:                         for $p \leftarrow 1$ to $InputWidth$ do
18:                             $I_{grad}[b, o_{ch}, row, p]$ +=
                                $\sum O_{grad}[b, o_{ch} : +24, r, p]*$
                                $W[o_{ch} : +n_{oc}, i_{ch} : +24, kh, :]$
19:                         end for
20:                     end for
21:                 end for
22:             end for
23:             Store $I_{grad}$ Row to Ext. Memory
24:         end for
25:     end for
26: end for
```

axis (lines 17-19), (2) the channel selection in the output gradient channel group is changed, while loading corresponding weight with processing output channel, input channel, and kernel height to the FMA trees (lines 13-15), (3) change the processing kernel height (line 12), (4) change the processing input channels (line 11) and store the finished input channel to external memory (line 23), (5) Load output gradient rows to the HP buffer (line 10), (6) change the active rows along the height axis of the input activation tensor (line 9), (7) change the current processing batch (line 8), and

(8) change the processing input channel split group, loading corresponding weights to HC buffer (lines 4-7). The global input routing unit, which consists of barrel shifters, 3-stage channel selection mux, and shift registers, aligns data according to its rows, channels, and spatial positions to allow this type of processing order. Although we dub this process *Conv-BP*, it should be noted that efficient implementations of this process are of interest to hardware designs only focusing on inference stages as well. The reason behind this is that the feedbackward processing of convolution is computationally identical to *deconvolution* (or also known as *transposed convolution*), which could be commonly found in generative models that makes use of CNNs.

Chapter 5

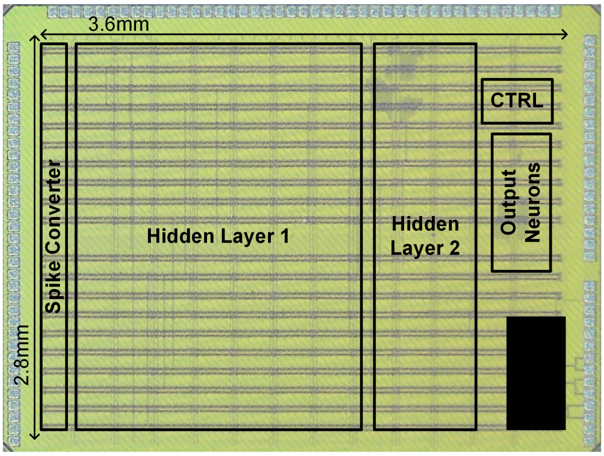
Measurement Results

5.1 Measurement Results on the Neuromorphic Learning System

In this section, the measurement results on the neuromorphic learning system is measured and analyzed. In particular, the effect of the update skipping mechanism is measured to compare with baseline design that does not implement this mechanism. Moreover, the system is compared against similar systems with on-chip learning for training energy overhead, as well as comparing the inference efficiency with various learning and inference-only systems.

5.1.1 Measurement Results and Test Setup

The neuromorphic learning system described in section 4.1 is fabricated in silicon and measured using custom test environment using PCBs and host FPGA environment built with Opal-Kelly FPGA boards. The die photograph of the fabricated integrated circuits is shown in Figure 5.1. This system is built using 65nm TSMC LPCMOS technology, a chip with 2.8mm×3.6mm core area and 353kB of on-chip memory implemented in technology vendor SRAM and register files. To verify this system, a test



©2020 IEEE

Figure 5.1: Die photo of the neuromorphic learning system.

environment is built using an FPGA enabled with USB 3.0 for control from host PC, communicating with the chip soldered on a custom PCB through general purpose I/O, with external voltage source and ampermeter connected to the core voltage source (VDD) to measure currents and power consumption. This test environment is shown in Figure 5.3. The general purpose I/O uses control sequence for interrupting or initiating training, changing the state of the learning system, and feeding raw MNIST image data at a speed of 32 bits per cycle. Power is measured through external current monitors from the core voltage source to the actual VDD used in the system, excluding the power of the test FPGA board. Clocks are given globally through a function generator, shared between the test FPGA and the design under test. Given this measurement setup, we develop high-level software kernel for communicating control logic to the neuromorphic system, retrieving monitored internal data of processor, receiving classified labels, and sending image pixel data to the neuromorphic processor based on a low-level software-hardware interface provided by OpalKelly FPGAs. Such kernels include *SystemReset*, *WriteMNISTImage*, *GetOutputPotentials*, *SetToInferenceMode*, *Interrupt*, etc. Block diagram of the design inside the test FPGA, are shown in Figure 5.2. Two operating modes are implemented in the test FPGA design for a full-speed

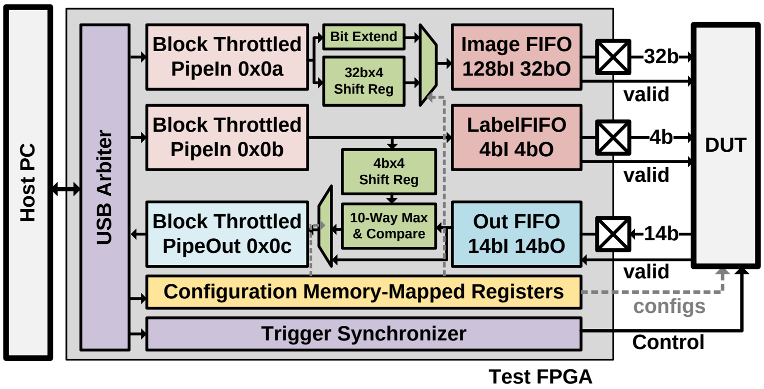


Figure 5.2: Diagram of the FPGA design for automated testing in the neuromorphic learning system.

verification mode, which automatically analyzes the output results in the test design and compares against the label data to record accuracy without interrupting the system.

At a maximum clock frequency of 50MHz with a nominal voltage of 1.2V, the system is trained with a throughput of 235.8K frames per second. At maximum energy efficiency of 20MHz with a nominal voltage of 0.8V, the system shows a training throughput of 94.3K frames while consuming 23.1mW. When converted to energy per image training, as is used in [25, 26], this result translates to 254.3nJ per image for training on MNIST dataset, and inference efficiency of 236.5nJ per image. As operations count for an MNIST image training in our algorithm is 804 kilo-Operations, we obtain 3.16TOPS/W of operating computational efficiency for training on MNIST images in our neuromorphic processor.

To illustrate the effect of the update skipping mechanism discussed in section 4.1.3, the training power throughout training for 60,000 images was measured using purposefully slow clock of 200KHz to capture training energy reduction in real time in the granularity offered by measuring equipment. We measure the training energy per image over the inference energy per image to evaluate the overeall overhead associated with training on our neuromorphic processor. In the first 1000 images, the energy required per training image relative to the inference energy was at 1.256 - this result is obtained using the inherent low overhead of training in the modified Segregated Dendrite algorithm, as this is a measured in the case where update skipping rates are very low (less than 3%). After learning progresses, the update skip rate is expected to increase, where the effect of update skipping mechanism kicks in. At the last 1000 images of 60000 images, the normalized energy required per training energy over inference energy was 1.082, which leads to the conclusion that the update skipping mechanism could reduce the training energy overhead from 25.6% to less than 8.2%. Over the average of 100 epochs that we trained the processor for, we measured an overhead of 7.5%. Therefore, we conclude that while the inherent low training overhead of the single-shot modified Segregated Dendrites algorithm provides us with low training energy overhead (25.6%) which still outperforms prior on-chip training works, the update skipping mechanism introduced in this work further reduces the training energy overhead from 25.6% to less than 1/3 (7.5%). This energy reduction graph and the update skipping rate is measured and plotted for training progression in a single epoch in Figure 5.4.

5.1.2 Comparison against other works

In order to compare the efficiency of our learning system against various learning systems, we compare the energy overhead of training normalized over inference energy. As inference takes 236.5nJ per image and training takes 254.3nJ per image, this learning system has a training energy overhead of only 7.5% while training from scratch to a classification accuracy of 97.83% on real time measurements, while prior works [39, 40, 41] with on-chip training showed a normalized training energy of 54.5% to 117.4%. Our low energy overhead was achieved mainly through two main implementation techniques: (1) Choice of energy-efficient algorithm. The modified segregated dendrites algorithm trains only using direct spiking feedback which only contains 20 bits of data in MNIST classification. (2) The update skipping mechanism which ex-

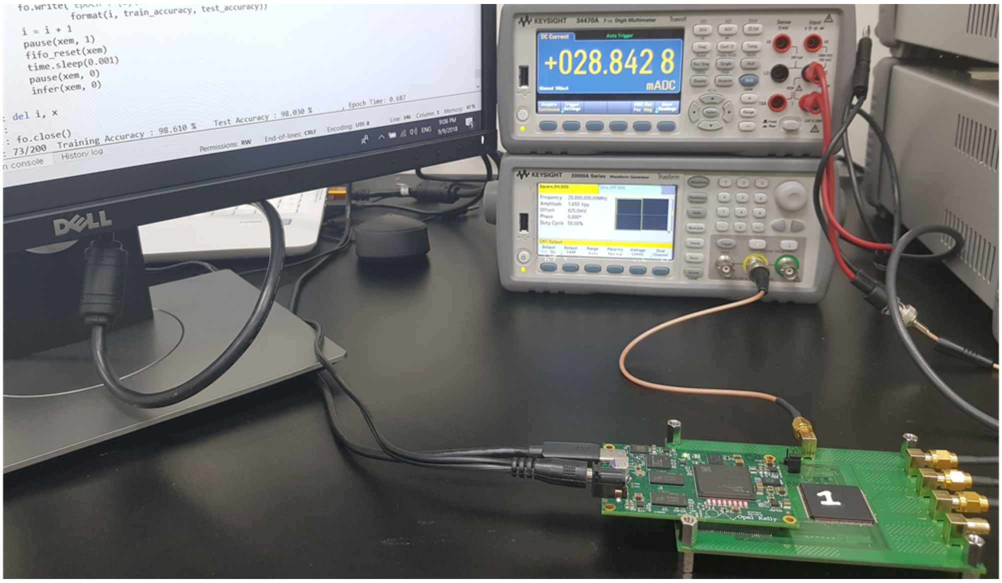


Figure 5.3: Test environment setup for real time measurement and verification of the fabricated learning system.

ploits sparsity of updates in the modified algorithm. Among those two factors, we conclude that the algorithm itself displays around 25.6% overhead for training (as this is the overhead when update skip ratio is close to zero), while the update skipping mechanism further brings this overhead down to 7.5%.

In order to validate the efficiency of the training in the implemented system, we used training energy normalized over inference as a metric. However, this metric is limited in the sense that it will look better for systems that have low inference efficiency: in this sense, it must always be accompanied by evaluation of the inference efficiency. Figure 5.5 shows a plot of energy consumption required for an MNIST image classification against the classification accuracy for this work and other SNN systems [25, 26, 42, 43, 44], with exception of [43] (shown in blue) that is a DNN inference-only chip. Our system improves the energy-accuracy trade-off for learning systems [25, 26, 41], even outperforming DNN inference-only chip with similar MNIST accuracy while consuming 34% less energy.

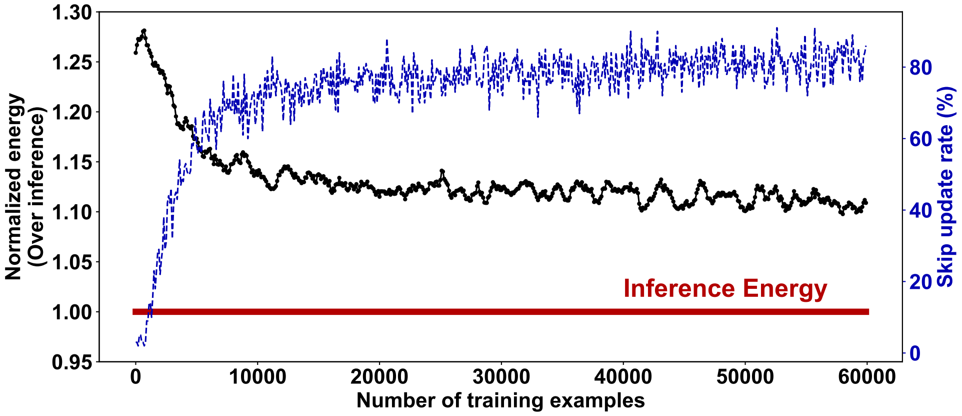
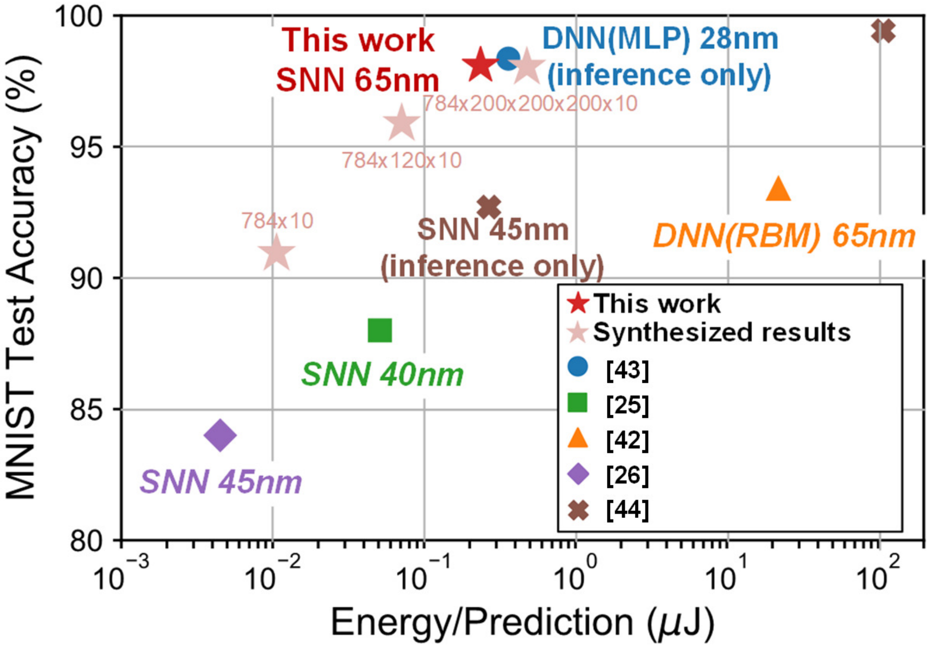


Figure 5.4: The update skipping rate and training energy normalized over inference energy is plotted.



©2020 IEEE

Figure 5.5: Plot of MNIST test accuracy and energy consumption per image for various learning systems.

Table 5.1: Energy Consumption of Neuromorphic System with Various Configurations*

Network Configuration	784-10	784-200-10	784-200-200-200-10
MNIST Accuracy	90.34%	96.4%	97.90%
Inference Energy	12.84nJ	87.87nJ	271.49nJ
Training Energy (without skip)	14.75nJ	107.36nJ	386.69nJ
Training Energy (with skip)	13.48nJ	93.11nJ	321.82nJ
Train Overhead (without skip)	14.8%	22.1%	42.4%
Train Overhead (with skip)	5.0%	6.0%	18.5%

*Synthesized results.

Moreover, to further show the energy-accuracy trade-off for on-chip learning systems, we synthesized and measured energy consumption using real test vectors for various network structures, plotting the energy and MNIST classification accuracy in Figure 5.5 and Table 5.1. This table better shows the accuracy-energy trade-off that is observed in various chips for MNIST image classifications, both inference and training. Putting this observation in perspective, the smallest network in Table 5.1 outperforms analog SNN system in [25] both in terms of energy (13.48nJ and 50.1nJ) and classification accuracy (90.34% and 88%), although it has to be considered that this is only a synthesized result. In Table 5.1, it is observed that training without update skipping shows overheads of 15% 42%, while training with update skipping shows overheads of 6% to 19% using real test vectors of the MNIST image dataset in our test environment, demonstrating the update skipping mechanism in various system and network configurations. This result also conforms to the observation in Figure 5.5 that there is an exponential energy trade-off for a linear increase in classification accuracy.

In summary, our design 34% less energy compared to state-of-art DNN inference chip on MNIST dataset during inference with 236.5nJ per image. For training, our system spends merely 7.5% more energy for performing training, much less overhead

Table 5.2: Comparison of On-Chip Learning Systems

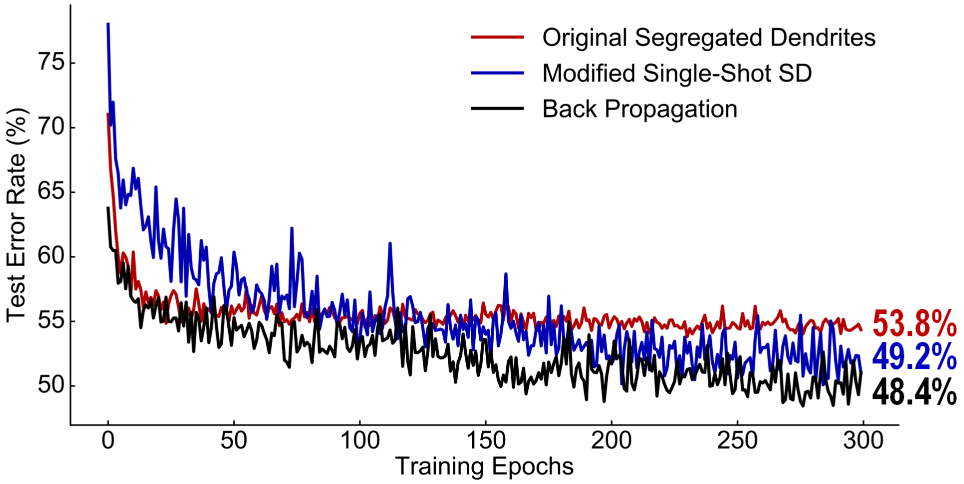
	Our Work [19]	[43]	[41]	[25]	[26]	[39]	[40]
Technology	**65nm**	28nm	65nm	40nm	65nm	65nm	55nm
On-Chip Training	**Yes**	No	Yes	Yes	Yes	Yes	Yes
MNIST Accuracy	**97.83%**	98.36%	93.4%	88%	84%	N/A	N/A
Algorithm	**SNN**	DNN(MLP)	DNN(RBM)	SNN	SNN	SVM	SVM
Prediction Energy (nJ/Image)	**236.5**	360	21900	50.1	4.5	N/A	N/A
Training Overhead	**7.5%***	N/A	54.5%	N/A	N/A	61.9%	117.4%
MNIST Throughput (FPS)	100K	15K	14K	2.2M	816K	N/A	N/A
Supply Voltage	0.8V	0.715V	1.2V	0.9V	0.425-0.45V	1.0V	0.4-1.0V
Frequency (MHz)	20-50	667-1200	210	250	20-310	1000	780

*Averaged over 100 training epochs.

compared to prior state-of-art that spent 54.5%, while showing an energy efficiency of 3.40TOPS/W. Moreover, our system delivers a latency of 12.7µs for inference with a throughput of 94.3K images per second for training, which is over 5× higher throughput for DNN training using Titan-X GPU on the same network configuration using back-propagation. Accounting for different convergence speeds of the algorithm, our network still manages to converge with less than a third of the time spent on training. These results are summarized and compared to other on-chip learning systems in Table 5.2.

5.1.3 Scalability of the Learning Algorithm

This work focused on implementing a relatively small size network for a fixed neuromorphic learning rule. However, the question of scalability of the algorithm remains – will this algorithm be scaled to larger, more complex tasks? For expanding the algorithm as-is using fully connected structures on CIFAR-10, it works moderately well for a fully connected network: While original SD algorithm achieves 46.2% accuracy on

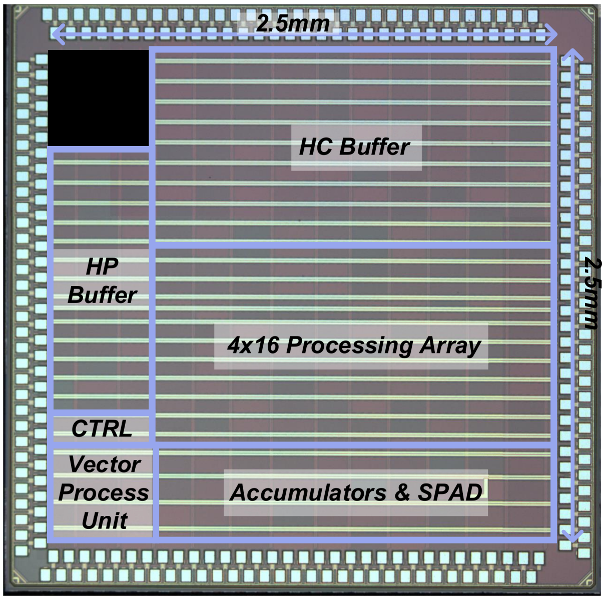


©2020 IEEE

Figure 5.6: Using fully connected network with size of 1024-512-10 for CIFAR-10 classification, trained with original SD, modified SD, and back propagation.

these CIFAR-10, our adapted SD algorithm achieves 50.8% accuracy, whereas network trained using back propagation showed 51.6% accuracy. This result is shown in Figure 5.6. During this training, update skip rate was measured to be at 31.5%. Similar model configuration using fully-connected networks on the CIFAR-100 dataset resulted in 16.2% accuracy with update skip rate of 3.2%, indicating the limited accuracy degrading update skip rates. In order to ensure higher performance in more complicated datasets, the segregated dendrites algorithm need to be modified for implementation in convolutional neural networks.

While our attempts at scaling single-shot modified Segregated Dendrites algorithm to convolutional neural networks were not successful, bio-plausible algorithms such as FA, DFA, and HFA were scaled to convolutional neural networks using methods that will be discussed in section 6.1. As the Segregated Dendrites could be considered as a close relative of the DFA algorithm family, we speculate similar methods could be utilized in this processor for scaling to CNNs in future works.



©2021 IEEE

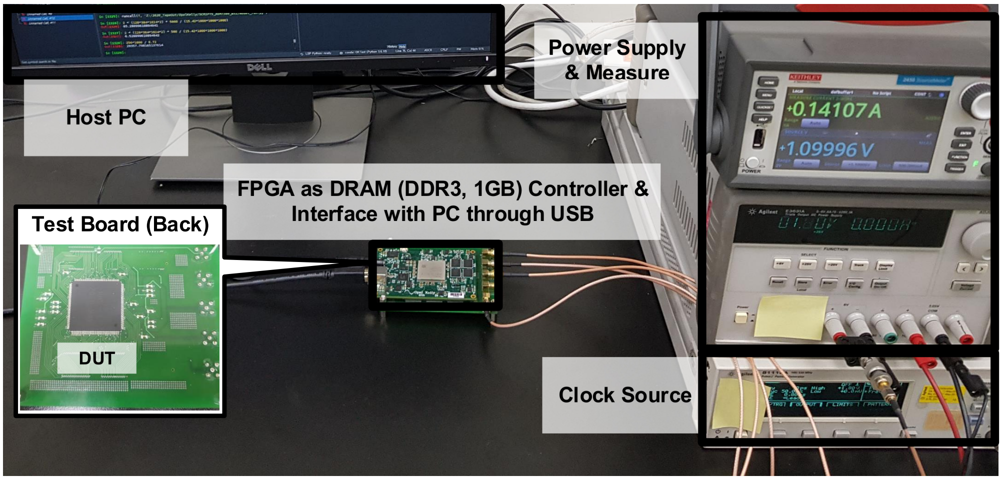
Figure 5.7: Die photograph of the fabricated DNN training processor.

5.2 Measurements Results on the Low-Precision DNN Training Processor

In this section, the measurement results on the DNN training processor utilizing FP8-SEB introduced in chapter 3 is measured and analyzed. Custom software chain is designed for automated testing of various benchmark models in real time. Secondly, the efficiency of the 2D input and output routing units introduced in this thesis are analyzed in both simulations and in measurements. Moreover, the fabricated processor's performance and power consumption is measured and analyzed on benchmark models including generative and language processing models. Finally, the efficiency of the DNN training processor is compared against other DNN training processors.

5.2.1 Measurement Results in Benchmarked Tests

We fabricated and verified the DNN training processor in TSMC 40nm LPCMOS technology. Figure 5.7 shows a micrograph of the 2.5mm×2.5mm core with 293kB



©2021 IEEE

Figure 5.8: Integrated test environment for automated measurement and verification of the DNN training processor.

of on-chip memory and 64 24-way FMA trees. In order to test, measure and verify the DNN training processor, we have also created an integrated test system with (1) a rudimentary design translator software that automatically creates binary files for running DNN models for training and inference on limited PyTorch models, (2) FPGA design with DDR3 memory controller available for direct memory access, granting access for both the host PC and the tested DNN training processor, (3) Processor control API in host PC, with requests sent to the FPGA through USB and to the tested DNN training processor for automated testing in real time, and finally (4) a designed and assembled PCB board connecting FPGA and DNN processor with 128-bit wide data bus and 32-bit wide address bus. This test system is shown in Figure 5.8.

Design Translator. We created a custom software toolchain that translates off-the-shelf PyTorch models to executable binary for our DNN training processor. First, the target model is parsed to intermediary representation (IR) using the default PyTorch's just-in-time (JIT) compiler in plain text. Next, the parsed IR is passed to a map-based model compiler that maps specific IR functions (e.g. batch normalization, convolution

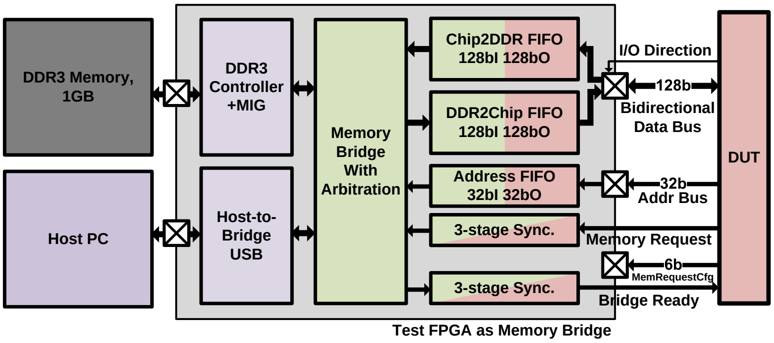


Figure 5.9: Diagram of the test FPGA serving as memory bridge.

cascaded with ReLU) to hand-designed instruction routines with pre-defined arithmetic precision as well as mapping basic arithmetic operations (e.g. +, -, x). Note that when creating executable binary for *training* phase of the model, this compiler also automatically maps backward and weight gradient functions associated with each arithmetic operation and functions, denoting the variables associated with training to generate the backward pass automatically. Finally, each variables are assigned address spaces and the instruction list is filled with the address space in place of the variables that are associated. It should be noted that this design translator currently hosts very limited selection of models, as the model compiler back-end is written with hand-designed maps for IR functions to DNN processor instructions and is currently defined for only limited number of IR functions. However, this still hosts enough compilation capabilities for the benchmark models in measured in this section.

FPGA Design. The FPGA acts as a memory bridge between three different components: (1) the DNN training processor (design under test), (2) the DDR3 SDRAM that is soldered on the FPGA board, and (3) host control PC connected through USB 3.0. In more detail, four asynchronous FIFO (first-in first-out) with 128-bit wide word and 1024 entrees aids data movement between the test components. As the DRAM is shared between DNN processor and the host PC, priority between the two needs to be established. While priority is given to DNN processor in principle, interrupt from host PC gives the host PC priority while ongoing data transaction in the DNN processor is stored for recovery until the interrupt is released. Using a system clock speed of 200MHz, the bridge design in the FPGA connects DDR3 DRAM memory to the DNN training processor's custom memory transfer protocol to deliver a maximum bandwidth of 3.2GB/s to the DNN training processor. The data bus is implemented in a bi-directional manner, with 3-stage synchronizer for request and ready signals in clock domain crossing. The block diagram of the host FPGA is shown in Figure 5.9.

Custom Memory Transfer Protocol. A custom data communication scheme using wide 128-bit I/O is implemented in the FPGA to be translated to the interfaces provided by FPGA vendor's memory interface generator. In this data communication scheme, all data transitions are made based on block-level transfer requests with transfer configurations packed in a 6-bit signal: (1) 4-bit number of words, which translates to 2^n number of transition words for a encoded number of words n, (2) 1-bit load-store select where signal high represents store mode, and finally (3) 1-bit mask that indicates bit-mask pattern is used in this transfer request: in this case, the first data crossing indicates the bit mask pattern in this transfer. These transfer configurations are held stable at the DNN training processor *before* sending memory request to the FPGA memory bridge. A memory transaction request is sent by an asynchronous request signal, passing through three-stage synchronizers to cross clock domains to the system clock in FPGA bridge. Similarly, the bridge indicates whether it is ready to accept new transactions. At the beginning of the transaction, configurable multi-cycle stalls are implemented for safe transition of the bi-directional bus to ensure no two conflicting drivers (one on the FPGA and another at the processor) drives the bus at the same time. More specifically, the select bit on each side of the bus is only able to be set to drive the bus if either (1) it has already been driving the bus (for example, the processor is *driving* the bus if it is in *write* phase, and if the next memory request is

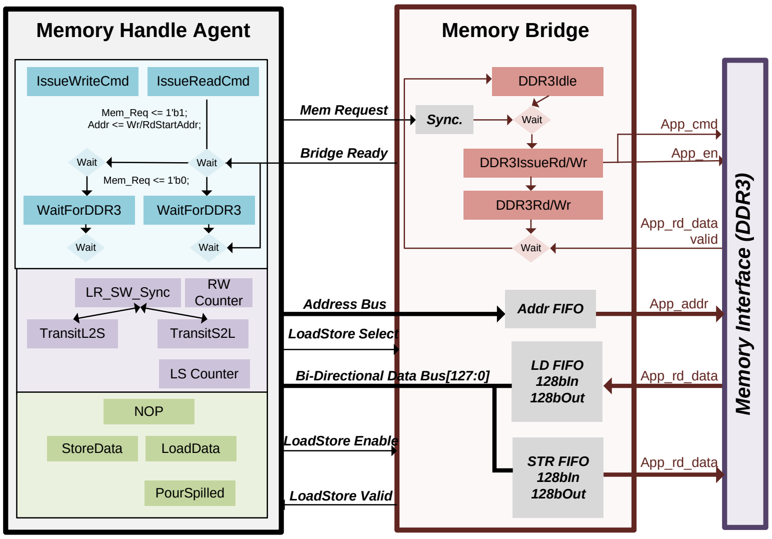


Figure 5.10: Block diagram of the custom memory transfer protocol.

also a write, this is the case of already driving the bus) or (2) it has not been driving the bus at the previous request, but is set to drive the bus at this request *and* the configurable stall cycles has passed since it indicated that it wishes to take control of the bus, allowing the other side driver to release control (hence being set to high-z state). After correct control of the bus has been established on both sides, transactions can begin with *LoadStoreValid* signal that indicates if the asynchronous FIFO has been able to proceed with the memory transaction that the DNN processor has requested. If this *LoadStoreValid* signal is high after an enable signal has been set from the processor, the counters on each side is incremented, repeating until the correct number of memory transactions has occurred. The block diagram of this memory protocol is shown in Figure 5.10.

Processor Control API. For automated testing and debugging of the fabricated DNN training processor, we developed high-level software routines with FPGA-host PC interface kernel provided by the FPGA vendor as the backend. For instance, *WriteTensor* routine is used for writing data to the DDR3 memory for the DNN training processor through the FPGA memory bridge, writing input data, parameters, and computation graphs (or *program*) required for DNN execution in the training processor. *CheckStates* routine is used to monitor the DNN training processor's states in real time, as well as the state of the memory bridge including FIFO counts. Other auxiliary routines such as *ExecuteGraph* helps execute the program multiple times for accurate measurement of power and latency.

PCB Design. For high-speed transaction of data between the test chip and the test, the printed circuit boards are designed carefully to avoid latency and skew incurred from implementation of the interconnect bus. 128-bit wide bus are designed such that (1) The longest routing path does not exceed length of 5cm and (2) largest skew between the shortest routing path and longest routing path does not exceed 2cm. Moreover, to prevent inter-cycle clock skew, global clock signal generated from pulse generators are implemented with impedance matching of 50 ohms. Through these efforts to reduce electromagnetic interference (EMI) from affecting signal integrity, a maximum operating frequency of 180MHz is achieved with standard LVCMOS I/O pads.

Based on the test environment described above, we measured the DNN training processor with varying operating points with different core voltage (VDD) and operating frequency to find optimal points in terms of performance and energy efficiency, as shown in Figure 5.11. Our DNN training processor operates up to 180MHz of core clock frequency at the nominal core voltage of 1.1V, and could operated at scaled down core voltage of 0.75V with 20MHz of core clock. Among the measured points, we analyze these two different operating points: (1) the first operating point with 180MHz clock and 1.1V core voltage is dubbed the *Maximum performance point*, while (2) the second operating point with 20MHz clock and 0.75V core voltage is dubbed the *maximum efficiency point*. Assuming 1 MAC operation as 2 FLOPs, the fabricated processor achieves 567 GFLOPS at maximum performance with 4.81 TFLOPS/W efficiency

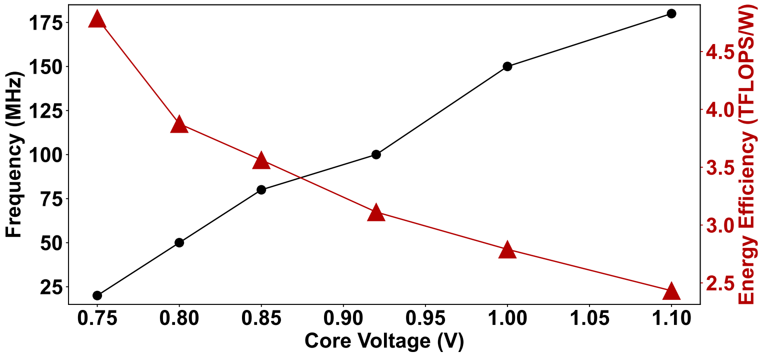


Figure 5.11: Performance-Efficiency trade-off graph with core voltage.

Table 5.3: Performance of the Processor on Benchmark Models

Model	ResNet-18		DC-GAN		LSTM	
Config	224×224 Image		512 to 32×32 Image		128×128 1-layer	
Train/Infer	Train	Infer	Train	Infer	Train	Infer
Energy Efficiency (TFLOPS/W)	1.64	2.05	1.63	2.05	0.31	0.34
Throughput (FPS)	27.2	92.1	220.5	766.0	82K	264K
EMA (bytes)	26.1M	5.2M	3.1M	0.6M	89.1K	39.9K
Number of MACs	5.46G	1.82G	0.63G	0.21G	0.32M	0.13M
MAC per Byte	209.2	350.0	203.2	350.0	3.6	3.3

at the maximm efficiency point.

To test our processor in more realistic DNN models, we benchmarked the processor on training and inference phase of three different network models: (1) ResNet-18 on ImageNet classification, (2) Single-layer LSTM for name classification, and (3) DC-GAN for handwritten digit generation. The benchmarked results are shown in Table 5.3. Note that *train* does not only include backward and parameter gradient generation phases, but also the feedforward path which are computationally very similar to inference phase. In the results shown, inference operations show higher energy efficiency due to two main reasons: (1) Auxiliary operations required for training generally require more memory-bound operations, such as weight updates and keeping track of weight momentums, which degrade OP/Byte ratio. (2) There is more space for optimization in forward phase, such as merging batch normalization with convolution layer. Note that inference showing higher efficiency is not due to conv-bp operations being inferior in terms of energy efficiency and utilization. This point is illustrated by the fact that the 'inference' phase of DC-GAN network is made up of transposed convolutions, which is essentially executed with conv-bp instructions. LSTMs show lower energy efficiency due to the operations mainly being bottlenecked at memory accesses, illustrated from the low MAC per byte ratio. The measurement results for measured EMAs and number of MACs in a model is also shown in Table 5.3, which demonstrates that the MAC per byte in training phase is lower compared against inference phases. The MAC per byte is calculated from the actual EMA usage, not from theoretical lower bound of external memory usage. In practical DNN training processors, theoretical lower bound memory accesses are difficult to achieve due to limited on-chip memory sizes.

To give more insight on how our processor spends its area and energy for operation, we use test vector inputs that were generated during actual measurement in a place-and-routed netlist. The power and area breakdown for a convolutional feedforward operation is shown in Figure 5.12. Accumulator and FMA trees, which are the main computing blocks, consume 50.3% of the total power, while on-chip memory (HP, HC, and scratchpads) consume 36.2% of total power. Note that the routing units proposed in chapter 4.2.3 take up very little energy and area with 0.72% and 0.34% of the total processor respectively. It should also be noted that reducing the accumulator precision does not only impact the hardware overhead in the accumulator (shown in medium

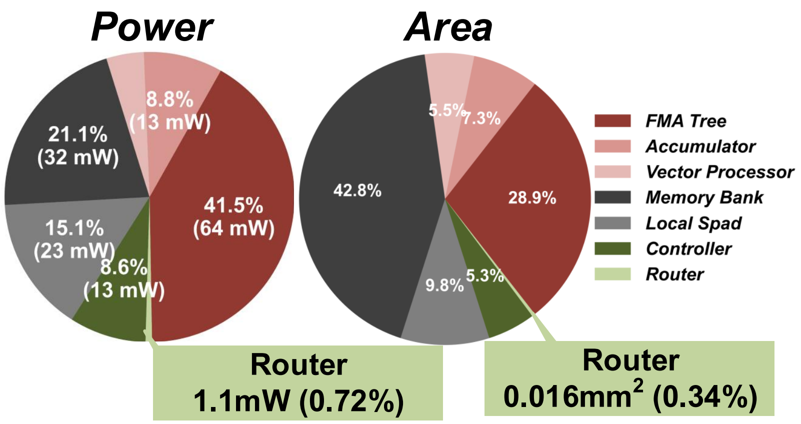


Figure 5.12: Area and power breakdown of the DNN training processor. Layer configuration is for conv-ff with 256 input channels, 256 output channels, 3×3 kernel size on a 14×14 image size.

light pink), but also reduces the cost of the FMA trees (red) and the local scratchpads (light gray). The reason for reduction in local scratchpads is because these scratchpads holds the same number of bits as the accumulation precision: a linear reduction is expected for reduction in accumulator precision. Complexity in the FMA trees are expected to be reduced as the representation length required for integer representation without precision loss in the accumulator is also reduced. For example, while 37-bit signed integers are required in the FMA trees for FP30 accumulator with (1.6.30) precision, only 24-bit signed integers are required for FMA trees with FP16 (1.6.9) accumulators. The reduction in hardware complexity with different accumulator bits are discussed in more detail in section 6.2.

Finally, to validate the spatial efficiency of the 2D input and routing units on the tree-based processing architecture, we compare the on-chip and off-chip data access counts required for processing a single model in each of the layers against another work [6] with spatial, systolic-array-like CNN accelerator. The data access counts are a good indicator of measuring the spatial efficiency, as the goal of spatial processing is to reduce redundant memory read and writes. By comparing the memory access required for convolution layers with the same configurations, we could quantitatively compare

Table 5.4: On-Chip and Off-Chip Memory Access in AlexNet Layers

Layer Types	**This Work**				**Eyeriss [6]**		
	On-Chip Buffer			Off-Chip	On-Chip Buffer		Off-Chip
	HP Buffer	HC Buffer	ScratchPad	DRAM	Global	ScratchPad*	DRAM
InCh=3, OutCh=64, Kernel=(11,11)	3.1MB	6.1MB	486.4MB	1.0MB	18.5MB	1938.6MB	5.0MB
InCh=64, OutCh=192, Kernel=(5,5)	10.7MB	7.0MB	320.4MB	0.3MB	77.6MB	4038.5MB	4.0MB
InCh=192, OutCh=384, Kernel=(3,3)	8.9MB	7.6MB	133.7MB	0.3MB	50.2MB	2594.6MB	3.0MB
InCh=384, OutCh=256, Kernel=(3,3)	11.9MB	10.1MB	178.2MB	0.6MB	37.4MB	1904.3MB	2.1MB
InCh=256, OutCh=256, Kernel=(3,3)	7.9MB	6.8MB	118.8MB	0.4MB	24.9MB	119.2MB	1.3MB
Total	**42.4MB**	**37.64MB**	**1238.5MB**	**2.6MB**	**208.5MB**	**11665.2MB**	**15.4MB**

*Inferred from original paper, using 2*16b*(# of Active PEs)*(# of Processing Cycles) as ScratchPad access count

the efficiency of spatial processing in the spatial architecture in [6] and the input-output routing tree-based architecture in our work. As the work in [6] reports memory access counts on AlexNet, we include the memory access counts required for Alexnet in Table 5.4. The advantage of using tree-based processing architecture is emphasised through the reduction of data access in the local scratchpads: our processor requires 89.3% less local scratchpad accesses (94.3% if data widths of partial sums are normalized to 16 bits). This reduction is *not* transferred to global bank accesses, as our 2D routing units on the input and output paths complement the tree-based processing architecture to maximize data reuse. This is supported by the fact that global buffer access for weights and input activations actually decrease by 61.6% (23.2% if widths of data access is normalized to 16 bits). Since the local scratchpads in [6] consumed 42.5% of the total power budget, we could conclude that tree-based processing architectures complemented with 2D routing paths may yield up to 40.2% less power just from the reduced accesses required for the local scratchpads. This data access reduction is achieved through the input and output routing units, as well as through the use of tree-based processing architecture. Moreover, this does not come at the cost of increased

Table 5.5: Comparison of Neural Network Training Processors

	Our Work [21]	[1]	[2]	[3]	[4]	[5]	[45]*
Technology	**40nm**	65nm	65nm	40nm	14nm	7nm	28nm
Data Format	**FP8-SEB**	FP8/FP16	FP8/FP16	BFLOAT16	FP16/FP32	HFP8	HBFP8
Peak Performance (GFLOPS)	**567**	300-600	540-1080	204	3000	25600	400000
Energy Efficiency (@sparsity=0%)	**4.81**	1.74-3.48	1.81-3.62	2.16	1.41	3.50	4.66
Real Model Efficiency	**1.64 (Res18)**	0.66-0.87 (Res18)	0.57-1.00 (CycGAN)	N/A	N/A	N/A	0.97 (LSTM) 0.21 (Res101)
On-Chip Memory	**293kB**	372kB	676kB	448kB	2MB	8MB	75MB
Core Area (mm^2)	**6.25**	16	32.4	16	9.24	19.6	314.0

*Synthesized Results

external memory access: only 2.6MB of external memory access is required against 15.4MB required in [6].

5.2.2 Comparison Against Other DNN Training Processors

This DNN training processor is compared against prior fabricated DNN training chips (with the exception of [45] which are simulated results from a synthesized netlist) in Table 5.5. During ResNet-18 training, our processor is measured at 1.64TFLOPS/W of energy efficiency which outperforms a prior work [1] with same ResNet-18 configuration by 2.48× under 0% sparsity condition, and with real sparsity conditions provided through ReLUs, still outperforms by 1.89×. Memory address generators and custom convolution logic through dedicated FSM control allows more efficient usage of on-chip memory for tiled convolution, requiring small on-chip memory (293kB) compared against similar works to achieve high computational density. This small on-chip memory does not result in excessive external memory accesses: in same ResNet-18 configuration, our processor require 43% less DRAM accesses.

To validate the training capabilities of the numerical format proposed in this DNN

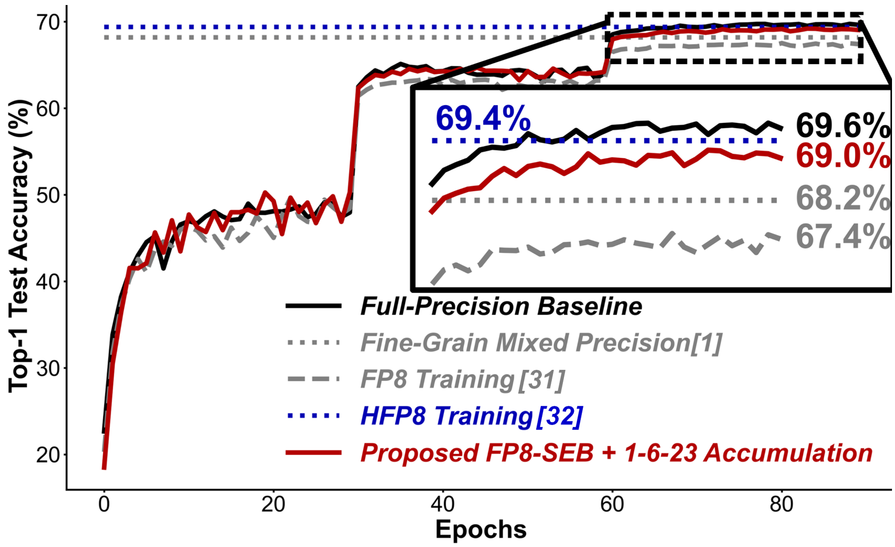


Figure 5.13: Training graph for different low-precision DNN training methods on ResNet-18 ImageNet classification benchmark.

processor, we compare our training performance on ResNet-18 Image Classification training with other low-precision works in Figure 5.13. We re-implemented the 8-bit training method proposed in [31] and used the reported figures from [1, 32]. Our FP8-SEB method outperforms fine grain mixed precision using FP8 and FP16 proposed in [1] despite only using 8 bits as inputs and outputs. With a top-1 classification accuracy of 69.0%, our results are comparable to trained top-1 accuracy of 69.39% reported by hybrid FP8 proposed in [32]. It should be noted that software simulations based on CUDA code that was verified to be bit-matched exactly with our DNN training processor was used to extract the training results on the ImageNet dataset in realistic amount time.

To evaluate the implementation costs of different FP8 formats introduced in other works [31, 32, 21], we implemented combinational adders, multipliers, and MAC units for 3 different FP8 formats, 1.4.3 FP8 (our format introduced in [21]), 1.5.2 FP8 [31], and Hybrid FP8 [32] implemented as 1.5.3 FP9. The results are shown in Table 5.6. We notice that in such tiny floating point representations, using more exponents is more

Table 5.6: Adders, multipliers, and MAC Units with Various FP8 Configurations.*

Logic Type		FP8(1.4.3)	FP8(1.5.2)	HFP8
Adder	Area(μm^2)	273.773	259.426	298.469
	Energy(pJ)	0.65	0.57	0.67
Multiplier	Area(μm^2)	408.307	354.92	455.35
	Energy(pJ)	0.755	0.61	0.84
MAC	Area(μm^2)	675.494	584.237	735.706
	Energy(pJ)	1.81	1.42	1.87

*Synthesis results in 40nm LP CMOS technology.

costly compared to using more mantissa bits. Note that in our HFP8 implementation, we excluded the cost of FP8-FP9 conversion units as they could be shared for arbitrary number of MAC units.

Finally, the cost of external memory should be included for comparison to give a more extensive view on the energy consumption of the DNN processor as DRAM is a major source of power consumption. It is well known that having larger on-chip memory sizes are more advantageous for reduced external memory access due to allowing larger tiled computations for reducing duplicate access to input/output data. Despite having smaller on-chip memory size against other works shown in Table 5.5, FP8-SEB tensor formats and optimized dataflow control for reduced memory access allows our processor to require only 21.6MB of external memory access, 43% less external memory access compared against the work that required 45.8MB in [1] for same ResNet-18 training benchmark. Note that for fair comparison, same DDR settings were used, constraining our processor to use only 256MB of the available DRAM, as more DRAM usage could benefit from larger batch sizes which could in turn mitigate the memory access overhead for fetching model parameters from the external memory.

In order to compare against GPU-based DNN training, we train ResNet-18 model on ImageNet dataset with a batch size of 64. Our training processor achieves energy efficiency of 1.64 TFLOPS/W, which is 78.1× improvement compared to GPU energy efficiency of 0.021 TFLOPS/W. In terms of DRAM usage, our processor requires only 573.9MB, 81.6% less compared to GPU DRAM consumption of 3127.0MB. This is still an improvement over the obvious savings from using 8-bit representations, which could theoretically save up to 75% over conventional full precision that utilizes 32 bits for representing tensor elements. We speculate this memory usage efficiency is a result of custom optimizations such as collecting garbage memory after it is freed from training process, specific optimized instruction control flow, and more compact code space for representing models.

Chapter 6

Conclusion

6.1 Discussion for Future Works

6.1.1 Scaling to CNNs in the Neuromorphic System

Measurement results of the Modified Segregated Dendrites algorithm in section 5.1.3 suggests that fully-connected networks that were implemented in our hardware are insufficient to perform on par with state-of-art DNN networks for real image processing. In order to fare better in real world applications, scaling the Modified Segregated Dendrites algorithm to convolutional neural networks may be a fascinating candidate for improvement.

However, the author was unable to directly scale the modified Segregated Dendrites algorithm to CNNs in many experiments, though more recent works in bio-plausible algorithms such as feedback alignment could shed the light in directions that could be explored for expanding the neuromorphic algorithm. For example, work in [27] and [28] suggests that alignment of angles in the feedforward and feedbackward weights in feedback alignment is the key to learning. In a similar manner, alignment angles of the DFA algorithm could be retrieved from computing the cosine similarity to an alignment target T_i, denoted by Equation 6.1, to the feedback matrix D_i in equation 2.17. The final alignment angle is computed as equation 6.2. Note that target

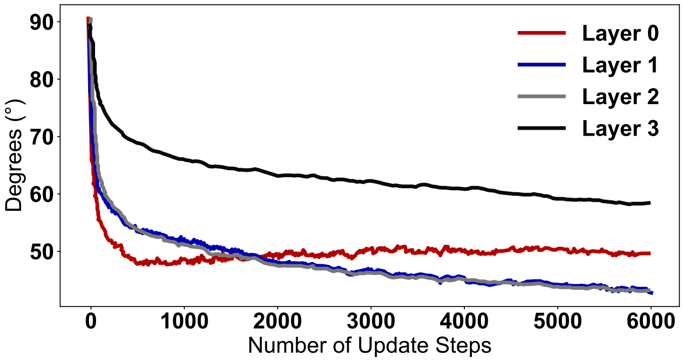


Figure 6.1: Alignment of each layers in a 4-layer fully-connected network to the target matrix in the modified Segregated Dendrites.

matrix changes as training progresses (as it is computed from feedforward weights) while the feedback matrix remains constant.

$$T_i = W_L W_{L-1} ... W_i \tag{6.1}$$

$$\theta_i = cos^{-1}(T_i \dot{D_i} / |T_i||D_i|) \tag{6.2}$$

To validate the claim of this comparison method, an experiment using 4-layer fully connected networks with the modified Segregated Dendrites algorithm to train on MNIST dataset is conducted. The angle computed by equation 6.2 is plotted for the training progression as in Figure 6.1. Alignment is observed between the target matrix and the feedback weigths, confirming the validity of our evaluation metric. Using our evaluation metric, we could not observe the alignment in CNN networks using the modified Segregated Dendrites rule, where convergence fails.

While this method could not be considered bio-plausible, method introduced in [27] could be applied for forcing the angle alignment between the target matrix and the feedback matrix. *Sign concordance*, which is the method introduced in the paper, suggests forcing alignments of feedback weights (R_i) to the target matrix using equation 6.3 at the end of every epoch for training.

$$R_i := sign(T_i) \tag{6.3}$$

Similarly, by bypassing the nonlinear functions that are placed in between convolution layers, we could merge convolution operators as they are linear operations. Suppose we are constructing the target matrix T for two cascaded convolution layers with weights W_1 and W_2. To obtain the equivalent convolution weight T, the intermediary convolution result $t[m, n]$ is shown in equation 6.4 and the final convolution result $y[m, n]$ is shown in equation 6.5, rearranged to equation 6.6.

$$t[m,n] = \sum_{i=0}^{k_1-1} \sum_{j=0}^{k_1-1} x[m+i, n+j]W_1[i,j] \tag{6.4}$$

$$y[m,n] = \sum_{l=0}^{k_2-1} \sum_{k=0}^{k_2-1} t[m+l, n+k]W_2[l,k] \tag{6.5}$$

$$y[m,n] = \sum_{l=0}^{k_2-1} \sum_{k=0}^{k_2-1} \sum_{i=0}^{k_1-1} \sum_{j=0}^{k_1-1} x[m+i+l][n+j+k]W_1[i,j]W_2[l,k] \tag{6.6}$$

By substituting $i' = i + l$ and $j' = j + k$, and letting $W_1[i, j] = 0$ for all out-of-bound indices, we obtain equation 6.7.

$$\begin{aligned} y[m,n] &= \sum_{l=0}^{k_2-1} \sum_{k=0}^{k_2-1} \sum_{i=0}^{k_1-1} \sum_{j=0}^{k_1-1} x[m+i'][n+j']W_1[i'-l, j'-k]W_2[l,k] \\ &= (\sum_{i'=0}^{k_1+k_2-2} \sum_{j'=0}^{k_1+k_2-2} x[m+i'][n+j'])(\sum_{l=0}^{k_2-1} \sum_{k=0}^{k_2-1} W_1[i'-l, j'-k]W_2[l,k]) \end{aligned} \tag{6.7}$$

By letting $W_{merge}[i', j']$ as the summation result of the second parenthesis in equation 6.7, we could see that this form is equivalent to convolution, concluding that the $W_{merge}[i', j']$ is the equivalent convolution weight. Thus, the target matrix T for convolution in DFA and HFA introduced in section 2.2 could be obtained using the following equation 6.8.

$$T[i,j] = \sum_{l=0}^{k_2-1} \sum_{k=0}^{k_2-1} W_1[i-l, j-k] W_2[l,k] \tag{6.8}$$

To test the validity of the target matrix proposed for convolutional neural networks in HFA algorithm introduced in section 2.2.1, we evaluated using the sign concordant algorithm [27] using equation 6.3 on MobileNet[52] structures to compare against naive HFA training without sign concordant. Similarly, using the sign concordant algorithm with the target matrix obtained through equation 6.8 could provide viable learning for modified Segregated Dendrites algorithm, and we leave this for future work.

6.1.2 Discussions for Improvements on DNN Training Processor

While the DNN training processor introduced in this thesis shows state-of-art results on some of the benchmarked models in terms of training energy efficiency, there still remains work to improve this processor for higher efficiency. We discuss three major directions for improvements that could benefit the current design in this section.

Use of lower precision in accumulation. Our training processor deploys 1.6.23 accumulators, as well as using lossless representations inside the FMA tree to ensure bit-match with GPU simulations of DNN training. However, by using FP16 (1.6.9) accumulators and allowing some loss in the FMA tree, our synthesized results showed that power consumption could be lowered in the computational units of the DNN training processor by approximately 20% less in total power consumption. A DNN training processor with FP16 accumulator is synthesized and analyzed on its average power consumption and area, with the results summarized in Table 6.1. As this method could not ensure bit match with current GPU simulation method, the training performance of this numerical pipeline needs to be analyzed. Through using FPGA-based simulations or building CUDA convolution kernels with lower accumulation precision, we could

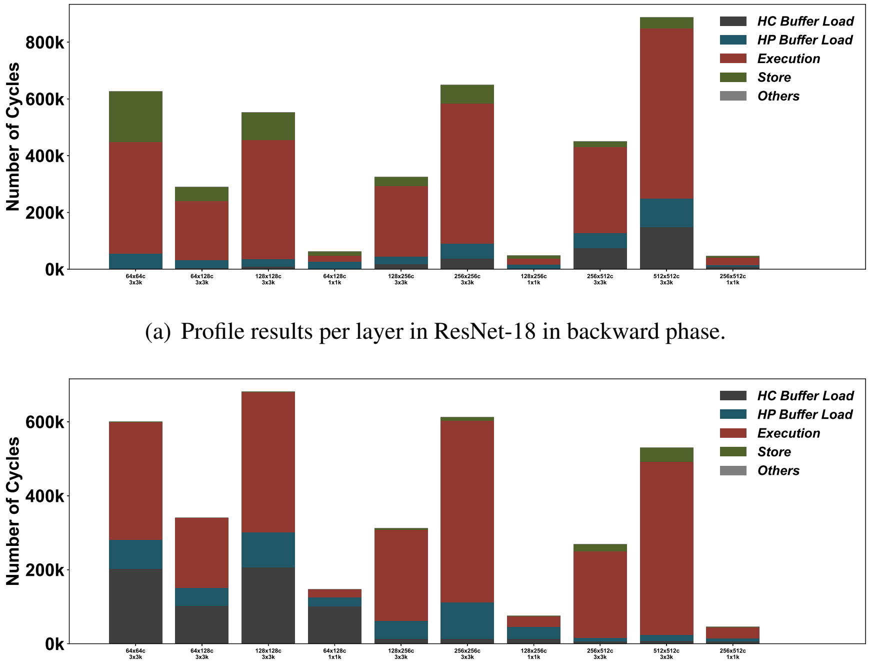
Table 6.1: DNN Training Processor Power and Area with FP16/FP30 Accumulators*

		DNN Processor with FP16 Accumulation	**DNN Processor with FP30 Accumulation [21]**
Power	FMA Tree	53.7mW	64.5mW
	Accumulator	7.9mW	13.7mW
	Local SPad	12.4mW	23.5mW
	Banks	28.2mW	29.0mW
	Others	18.2mW	18.9mW
	Total	**120.4mW**	**149.6mW**
Area	Seq. and Memory	2.64mm^2	2.97mm^2
	Combinational	1.60mm^2	1.75mm^2
	Total	**4.24mm^2**	**4.69mm^2**

*Synthesized results.

evaluate the impact of reduced accumulation precision for lowered power consumption in future designs.

Mitigating memory latency. Another aspect that could be improved in current DNN training processor is mitigating memory latencies. Although in chapter 4.2.2 we discuss memory prefetchers for HP buffers, we could not mitigate the latency from loading network parameters to the HC buffer in the current design. Prefetchers for HC buffer would help mitigate this in future designs. Figure 6.2(a) and Figure 6.2(b) shows the breakdown of number of cycles per processor states for Resnet-18 backward and weight gradient phases respectively. Exclusion of prefetching mechanism for load and store of gradients incur more overhead in HP buffer loading with 9.1% of total execution time, contrary to HP buffer loading taking up 5.6% of total execution time during feedforward operations. Moreover, use of L1-L2 caches could also help reduce memory latencies especially for memory-bound operations such as batch normalization, although this was not included in our current design due to area constraints.



(a) Profile results per layer in ResNet-18 in backward phase.

(b) Profile results per layer in ResNet-18 in weight gradient phase.

Figure 6.2: Processor state breakdown per layer in the backward and weight gradient phase of ResNet-18 training.

Optimization for Inference. Moreover, this DNN training processor could be extended to handle inference workloads more efficiently. Processors then can be utilized for server-scale DNN workload accelerators to be used for both inference and training, even allowing training to be processed while responding to inference task requests, similar to work in [45]. For example, hybrid mode FMA trees could be implemented such that one FMA tree performing vector FMA between (1.4.3)×(1.4.3) formats could also be configured to act as two FMA trees each performing vector FMA between (1.4.3)×(1.3.0) formats, which showed robust inference performance.

6.2 Conclusion

In this dissertation, various techniques for designing low power neural network training integrated circuits system are introduced, in terms of training algorithms, reduced numerical precision, and digital circuit implementation techniques. Specifically, two fabricated chips were designed, verified and measured for evaluation of these techniques. The first chip demonstrated a neuromorphic learning system with very low training energy overhead. The second chip implements a custom FP8 numeric format for energy-efficient training, with specialized instruction set for end-to-end DNN training in spatial processing architecture. In conclusion, the contributions of this thesis could be summarized as following:

Neuromorphic Algorithm Modification and Update Skipping In the neuromorphic learning system, we modify an existing learning rule [20] for more hardware-efficient implementation. The modifications reduce operations required per training from 29.4MOP to 0.8MOP, as well as reducing the buffer memory requirement from 42.7Kb to 7.6Kb, while increasing the MNIST test accuracy from 96.3% to 98.1%. Moreover, the sparsity of the learning rule is exploited in hardware to further reduce training energy overhead from 25.6% of the inference energy to a mere 7.5%.

FP8 with Shared Exponent Bias The DNN training processor chip introduced in this thesis is implemented with a custom FP8 format, which we dubbed FP8-SEB (shared exponent bias). The shared exponents allow a inter-tensor dynamic range that matches full-precision, while maintaining intra-tensor dynamic range of 44.9dB. While accessing off-chip tensors in only 8-bit representations, our numeric format matches full-precision baseline training in various benchmarks, including tasks in image classification (ImageNet classification using ResNet-18 [30]), generative task (Image Super-resolution using ESRGAN [47]), and natural language processing (Image Captioning using attention mechanisms and LSTMs [48]).

Flexible Routing Scheme in Tree-based Processing Architecture We obtain more energy-efficient processing elements through the use of multiple way fused-multiply-adder trees that consumes 87% less energy compared to straightforward MAC-based implementation. While other works have also proposed using efficient tree-based processing architecture, limited spatiality and data re-use has limited the energy efficiency of such processing architectures. Our novel 2D routing schemes on both inputs and outputs outperforms prior spatial processing architecture based on systolic arrays while offering very low hardware overhead that takes up 0.72% of total power and 0.34% of total area.

Extending Bio-Plausible learning rules to DNN training processor Another contribution of this work is verifying the hardware efficiency of bio-plausible learning rules such as FA, DFA, and our proposed HFA. Through the use of on-chip random weight generation and gradient sharing, proposed HFA algorithm could perform up to 23% faster and with 21% lower energy in our DNN training processor, with the hardware advantages shown to persist in computational scaling to settings of training on ImageNet data. Moreover, this efficiency could be utilized with very low software overhead through using mixed training schema of stochastically choosing either back-propagation or the HFA learning rule on the processing mini-batch.

Bibliography

[1] J. Lee, J. Lee, D. Han, J. Lee, G. Park, and H.-J. Yoo. LNPU: A 25.3 TFLOPS/W sparse deep-neural-network learning processor with fine-grained mixed precision of FP8-FP16. In *IEEE International Solid-States Circuits Conference (ISSCC)*, pages 142–144. IEEE, 2019.

[2] S. Kang, D. Han, J. Lee, D. Im, S. Kim, S. Kim, and H.-J. Yoo. GANPU: A 135TFLOPS/W multi-DNN training processor for GANs with speculative dual-sparsity exploitation. In *IEEE International Solid-States Circuits Conference (ISSCC)*, pages 140–142, 2020.

[3] C. Kim, S. Kang, D. Shin, S. Choi, Y. Kim, and H.-J. Yoo. A 2.1 TFLOPS/W mobile deep RL accelerator with transposable PE array and experience compression. In *IEEE International Solid-States Circuits Conference (ISSCC)*, pages 136–138, 2019.

[4] B. Fleischer, S. Shukla, M. Ziegler, J. Silberman, J. Oh, V. Srinivasan, J. Choi, S. Mueller, A. Agrawal, and T. Babinsky. A scalable multi-TeraOPs deep learning processor core for AI training and inference. In *IEEE Symposium on Very Large Scale Integrated Circuits (VLSIC)*, pages 35–36, 2018.

[5] A. Agrawal, S. K. Lee, J. Silberman, M. Ziegler,M. Kang, S. Venkataramani, N. Cao, B. Fleischer, M. Guillorn, and M. Cohen. A 7nm 4-core AI chip with 25.6 TFLOPS hybrid FP8 training, 102.4 TOPS INT4 inference and workload-aware

throttling. In *IEEE International Solid-States Circuits Conference (ISSCC)*, pages 144–146, 2021.

[6] Y. Chen, T. Krishna, J. Emer, and V. Sze. Eyeriss:An energy-efficient reconfigurable accelerator for deep convolutional neural networks. In *IEEE Journal of Solid-State Circuits*, volume 52, pages 127–138, 2016.

[7] J. Song, Y. Cho, J.-S. Park, J.-W. Jang, S. Lee, J.-H. Song, J.-G. Lee, and I. Kang. An 11.5 TOPS/W 1024-MAC butterfly structure dual-core sparsity-aware neural processing unit in 8nm flagship mobile SoC. In *IEEE International Solid-States Circuits Conference (ISSCC)*, pages 130–132, 2019.

[8] S. Kim, J. Lee, S. Kang, J. Lee, and H.-J. Yoo. A 146.52 TOPS/W deep-neural-network learning processor with stochastic coarse-fine pruning and adaptive input/output/weight skipping In *IEEE Symposium on Very Large Scale Integrated Circuits (VLSIC)*, pages 1–2, 2020.

[9] J. Oh, S. K. Lee, M. Kang, M. Ziegler, J. Silberman,A. Agrawal, S. Venkataramani, B. Fleischer, M. Guillorn, and J. Choi. A 3.0 TFLOPS 0.62 v scalable processor core for high compute utilization AI training and inference In *IEEE Symposium on Very Large Scale Integrated Circuits (VLSIC)*, pages 1–2, 2020.

[10] P. Jouppi *et al.* In-datacenter performance analysis of a tensor processing unit In *Proceedings of the 44th annual International Symposium on Computer Architecture (ISCA)*, pages 1–12, 2017.

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

[12] J. Konečný, H. B. McMahan, F. X. Yu, P. Richtárik, A. T.Suresh, and D. Bacon. Federated learning: Strategies for improving communication efficiency *arXiv preprint arXiv:1610.05492*, 2016.

[13] R. Krishnamoorthi. Quantizing deep convolutional networks for efficient inference: A whitepaper *arXiv preprint arXiv:1806.08342*, 2018.

[14] M. Rastegari, V.Ordonez, J. Redmon, and A.Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision (ECCV)*, Springer, 2016.

[15] S. Gupta, A. Agrawal, K. Goplakrishnan, and P. Narayanan. Deep learning with limited numerical precision. *arXiv preprint arXiv:1502.02551*, 2015.

[16] 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:1606.06160*, 2016.

[17] U. Köster *et al.* Flexpoint: An adaptive numerical format for efficient training of deep neural networks *arXiv preprint arXiv:1711.02213*, 2017.

[18] J. Eshraghian, M. Ward, E. Neftci, X. Wang, G. Lenz, G. Dwivedi, M. Bennamoun, D. Jeong, and W.Lu. Training spiking neural networks using lessons from deep learning *arXiv preprint arXiv:2109.12894*, 2021.

[19] J. Park, J. Lee, and D. Jeon. A 65nm 236.5nJ/classification neuromorphic processor with 7.5% Energy overhead on-chip Learning using direct spike-only feedback In *IEEE International Solid-States Circuits Conference (ISSCC)*, pages 140–141, 2019.

[20] J. Guerguiev, T. Lillicrap, and B. Richards. Towards deep learning with segregated dendrites. In *ELife*, vol. 6, 2017.

[21] J. Park, S. Lee, and D. Jeon. A 40nm 4.81 TFLOPS/W 8b floating-point training processor for non-sparse neural networks using shared exponent bias and 24-way fused multiply-add tree. In *IEEE International Solid-States Circuits Conference (ISSCC)*, pages 1–3, 2021.

[22] H, Markram, G. Wulfram , and S. Jesper. A history of spike-timing-dependent plasticity. In *Frontiers in Synaptic Neuroscience*, vol. 3, 2011.

[23] T. Lillicrap, D. Cownden, D. Tweed, and C. Akerman. Random synaptic feedback weights support error backpropagation for deep learning. In *Nature Communications*, vol. 7, 2016.

[24] A. Nøkland. Direct feedback alignment provides learning in deep neural networks. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2016.

[25] F. Buhler, P. Brown, J. Li, T. Chen, Z.Zhang, and M. Flynn. A 3.43TOPS/W 48.9pJ/pixel 50.1nJ/classification 512 analog neuron sparse coding neural network with on-chip learning and classification in 40nm CMOS. In *IEEE Symposium on Very Large Scale Integrated Circuits (VLSIC)*, pages 30–31, 2017.

[26] J. Kim, P. Knag, T. Chen, and Z. Zhang. A 640M pixel/s 3.65mW sparse event-driven neuromorphic object recognition processor with on-chip Learning. In *IEEE Symposium on Very Large Scale Integrated Circuits (VLSIC)*, pages 61–62, 2015.

[27] T. Moskovitz, L. Ashok, and L. Abbott. Feedback alignment in deep convolutional networks. *arXiv preprint arXiv:1812.06488*, 2018.

[28] W. Xiao, H. Chen, Q. Liao, and T. Poggio. Biologically-plausible learning algorithms can scale to large datasets. *arXiv preprint arXiv:1811.03567*, 2018.

[29] L. Min, Q. Chen, and S. Yan. Network in network. *arXiv preprint arXiv:1312.4400*, 2013.

[30] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778, 2016.

[31] N. Wang, J. Choi, D. Brand, C.-Y. Chen, and K. Gopalakrishnan. Training deep neural networks with 8-bit floating point numbers. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2018.

[32] X. Sun, J. Choi, C. Y. Chen, N. Wang, S. Venkataramani, V. Srinivasan, X. Cui, W. Zhang, and K. Gopalakrishnan. Hybrid 8-bit floating point (hfp8) training and inference for deep neural networks In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 1—10, 2019.

[33] M. Drummond, T. Lin, and M. Jaggi, and B. Falsafi. Training DNNs with hybrid block floating point. *arXiv preprint arXiv:1804.01526*, 2018.

[34] S. Fox *et al.* A Block minifloat representation for training deep neural networks. In *International Conference on Learning Representations (ICLR)*, 2020.

[35] B. Xu, N. Wang, T. Chen and M. Li. Empirical evaluation of rectified activations in convolution Network. *arXiv preprint arXiv:1505.00853*, 2015.

[36] F. Niu, B. Recht, C. Ré, and S. Wright. Hogwild!: A lock-free approach to parallelizing stochastic gradient descent *arXiv preprint arXiv:1106.5730*, 2011.

[37] J. Tsitsiklis, B. Dimitri, and M. Athans. Distributed asynchronous deterministic and stochastic gradient optimization algorithms. In *IEEE transactions on automatic control*, pages 803–812, 1986.

[38] J. Park *et al.* A 6K-MAC Feature-map-sparsity-aware neural processing unit in 5nm flagship mobile SoC. In *IEEE International Solid-States Circuits Conference (ISSCC)*, pages 152–154, 2021.

[39] S. Gonugondla, S. Kumar, M. Kang, and N. Shanbhag. A 42pJ/decision 3.12TOPS/W roust in-memory machine learning classifier with on-chip training. In *IEEE International Solid-States Circuits Conference (ISSCC)*, pages 490–491, 2018.

[40] A. Amravati, S. Nasir,i S. Thangadurai, I. Yoon, and A. Raychowdhury. A 55nm time-domain mixed-signal neuromorphic accelerator with stochastic synapses and embedded reinforcement learning for autonomous micro-robots. In *IEEE International Solid-States Circuits Conference (ISSCC)*, pages 124–125, 2018.

[41] C. Tsai, W. Yu, W. Wong, and C. Lee. A 41.3/26.7pJ per neuron weight RBM processor supporting on-chip learning/inference for IoT applications. In *IEEE Journal of Solid-State Circuits*, volume 52, pages 2601-2612, 2017.

[42] J. Seo *et al.* A 45nm CMOS neuromorphic chip with a scalable architecture for learning in networks of spiking neurons. In *IEEE Custom Integrated Circuits Conference (CICC)*, pages 1–4, 2011.

[43] P. Whatmough, S. Lee, H. Lee, S. Rama, D. Brooks, and G. Wei. A 28nm SoC with a 1.2GHz 568nJ/prediction sparse deep-neural-network engine with ¿ 0.1 timing error rate tolerance for IoT applications In *IEEE International Solid-States Circuits Conference (ISSCC)*, pages 242–243, 2017.

[44] S. Esser, R. Appuswamy, P. Merolla, J. Arthur, and D. Modha. Back propagation for energy-efficient neuromorphic computing. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2015.

[45] M. Drumond, L. Coulon, A. Pourhabibi, A. Yüzügüler, B. Falsafi and M. Jaggi. Equinox: Training (for Free) on a custom inference accelerator In *54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 421–433, 2021.

[46] P. Merolla *et al.* A million spiking-neuron integrated circuit with a scalable communication network and interface. *Science*, pages 668–673, 2014.

[47] X. Wang, K. Yu, S. Wu, J. Gu, Y. Liu, C. Dong, Y. Qiao, C. Loy. ESRGAN: enhanced super-resolution generative adversarial networks. In *European Conference on Computer Vision (ECCV)*, Springer, 2018.

[48] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. Gomez, L. Kaiser, and I. Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

[49] A. Paszke *et al.* PyTorch: An imperative style, high-performance deep learning library In *Advances in Neural Information Processing Systems (NeurIPS)*, 2019.

[50] W. Gerstner and W. Kistler. Spiking neuron models : single neurons, populations, plasticity *Cambridge University Press*, 2002.

[51] A. Hodgkin and A. Huxley. A quantitative description of membrane current and its application to conduction and excitation in nerve. In *The Journal of Physiology*, pages 500–544, 1952.

[52] A. 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.

초 록

딥러닝의 시대가 도래함에 따라, 심층 인공 신경망 (DNN)을 처리하기 위해 요구되는 학습 및 추론 연산량 또한 기하급수적으로 증가하였다. 딥 러닝 시대의 도래와 함께 다양한 작업에 대한 신경망 훈련 및 특정 용도에 대해 훈련된 신경망 추론 수행 측면에서 심층 신경망 (DNN) 처리에 대한 컴퓨팅 요구가 극적으로 증가하였으며, 이러한 추세는 인공지능의 사용이 더욱 범용적으로 진화함에 따라 더욱 가속화 될 것으로 예상된다. 이러한 연산 요구를 해결하기 위해 데이터 센터 내부에 배치하기 위한 FPGA (Field-Programmable Gate Array) 또는 ASIC (Application-Specific Integrated Circuit) 기반 시스템에서 저전력을 위한 SoC (System-on-Chip)의 가속 블록에 이르기까지 다양한 맞춤형 하드웨어가 산업 및 학계에서 제안되었다. 본 논문에서는, 인공 신경망의 에너지 효율적인 훈련 처리를 위한 맞춤형 집적 회로 하드웨어를 보다 에너지 효율적으로 설계할 수 있는 다양한 방법론을 제안하고 실제 저전력 인공 신경망 훈련 시스템을 설계하고 제작하여, 그 효율을 평가하고자 한다. 특히, 본 논문에서는 이러한 저전력 고성능 설계 방법론을 크게 세 가지로 분류하여 분석을 진행하였다. 이러한 분류는 다음과 같다. (1) **훈련 알고리즘**. 표준적으로 심층 신경망 훈련은 역전파 (Back-Propagation) 알고리즘으로 수행되지만, 더 효율적인 하드웨어 구현을 위해 스파이크을 기반으로 통신하는 뉴런이 있는 뉴로모픽 학습 알고리즘 또는 비대칭 피드백 을 기반으로 하는 생물학적 모사도가 높은 (Bio-Plausible) 알고리즘을 활용하여 더 효율적인 훈련 시스템을 설계하는 방법을 조사 및 제시하고, 그 하드웨어 효율성을 분석하였다. (2) **저정밀도 수 체계 활용**. 일반적으로 사용되는 DNN 가속기에서 효율성을 높이는 가장 강력한 방법 중 하나는 수치 정밀도를 조정하는 것이다. DNN의 추론 단계에 낮은 정밀도 숫자를 사용하는 것은

잘 연구되었지만, 성능 저하 없이 DNN을 훈련하는 것은 상대적으 기술적 어려움이 있다. 본 논문에서는 다양한 모델과 시나리오에서 DNN을 성능 저하 없이 훈련하기 위한 새로운 수 체계를 제안하였다. (3) **시스템 구현 기법**. 집적 회로에서 맞춤형 훈련 시스템을 실제로 실현할 때, 거의 무한한 설계 공간은 칩 내부의 데이터 흐름, 시스템 부하 분산, 가속/게이팅 블록 등 다양한 요소에 따라 결과의 품질이 크게 달라질 수 있다. 본 논문에서는 더 나은 성능과 효율성으로 이어지는 다양한 설계 기법을 소개하고 분석하고자 한다.

첫째로, 손글씨 분류 학습을 위한 뉴로모픽 학습 시스템을 제작하여 평가하였다. 이 학습 시스템은 전통적인 기계 학습의 훈련 성능을 유지하면서 낮은 훈련 오버헤드를 제공하는 것을 목표로 하여 설계되었다. 이 목적을 달성하기 위해, 더 적은 연산 요구량과 버퍼 메모리 필요치를 위해 기존의 뉴로모픽 알고리즘을 수정하였으며, 이 과정에서 훈련 성능 손실 없이 기존 역전파 기반 알고리즘에 근접한 훈련 성능을 달성하였다. 뿐만 아니라, 업데이트를 건너뛰는 메커니즘을 구현하고 Lock-Free 매개변수 업데이트 방식을 채택하여 훈련에 소모되는 에너지를 훈련이 진행됨에 따라 동적으로 감소시킬 수 있는 시스템 구현 기법 또한 소개하고 그 성능을 분석하였다. 이런 기법을 통해, 이 학습 시스템은 기존의 훈련 시스템 대비 뛰어난 분류 성능-에너지 소모량 관계를 보이면서도 기존의 역전파 알고리즘 기반의 인공 신경망의 훈련 성능을 유지하였다.

둘째로, 특수 명령어 체계 및 맞춤형 수 체계를 활용한 프로그램 가능한 DNN 훈련용 프로세서가 설계되고 제작되었다. 기존 DNN 추론용 가속기는 8비트 정수 기반으로 이루어진 경우가 많았지만, DNN 학습 설계시 8비트 수 체계를 이용하며 훈련 성능 저하를 보이지 않는 것은 상당한 기술적 난이도를 가지고 있었다. 이런 문제를 극복하기 위해, 본 논문에서는 공유형 멱지수 편향값을 활용하는 8비트 부동 소수점 수 체계를 새로이 제안하였으며, 이 수 체계의 효용성을 보이기 위해 이 DNN 훈련 프로세서가 설계되었다. 뿐만 아니라, 이 프로세서는 단순한 MAC 기반 Matrix-Multiplication 가속기가 아닌, Fused-Multiply-Add 트리를 기반으로 하는 에너지 효율적인 가속기 구조를 채택하면서도, 칩 내부에서의 데이터 이동량 최적화 및 컨볼루션의 공간성을 극대화할 수 있기 위해 데이터 전달 유닛을 입출력부에 2D

로 제작하여 트리 기반에서의 컨볼루션 추론 및 훈련 단계에서의 공간성을 활용할 수 있는 방법을 제시하였다. 본 DNN 훈련 프로세서는 맞춤형 벡터 연산기, 가속 명령어 체계, 외부 DRAM으로의 직접적인 접근 제어 방식 등을 통해 한 프로세서 내에서 DNN 훈련의 모든 단계를 다양한 모델 및 환경에서 효율적으로 처리할 수 있도록 설계되었다. 이를 통해 본 프로세서는 기존의 연구에서 제시되었던 다른 프로세서에 비해 동일 모델을 처리하면서 2.48배 가량 더 높은 에너지 효율성, 43% 적은 DRAM 접근 요구량, 0.8%p 높은 훈련 성능을 달성하였다.

이렇게 소개된 두 가지 설계는 모두 실제 칩으로 제작되어 검증되었다. 측정 데이터 및 전력 소모량을 통해 본 논문에서 제안된 저전력 딥러닝 훈련 시스템 설계 기법의 효율을 검증하였으며, 특히 생물학적 모사도가 높은 훈련 알고리즘, 딥러닝 훈련에 최적화된 수 체계, 그리고 효율적인 시스템 구현 기법을 활용하여 시스템의 에너지 효율성을 개선하는 목표를 달성하였는지 정량적으로 분석하였다.

주요어: 인공 신경망 학습 가속기, 디지털 집적회로, 딥 러닝, 연산 효율, 학습 시스템, 뉴로모픽, 기계학습 가속기

학번: 2017-22051