Hardware Accelerators for Deep Neural Networks

by

Raju Machupalli

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

in
Computer Engineering

Department of Electrical and Computer Engineering
University of Alberta

Abstract

Deep Neural Networks (DNNs) have recently evolved as the state-of-the-art method for machine learning applications such as object detection, face recognition, and image classification. However, a DNN typically has high computational complexity, and specialized hardware accelerators would be helpful to obtain real-time performance.

Over the last decade, many accelerators have been proposed in the literature for DNN models. This thesis presents a comprehensive review of the existing DNN accelerators. The accelerators were classified into four categories: ALU, Dataflow, Sparsity, and Hybrid, based on the optimization techniques used. The classification provides a good starting point to identify significant areas where an accelerator can be further optimized for better throughput, latency, and energy performance.

In this thesis, we also explored the bit-precision requirement of the MAC units for DNN implementation. A DNN has two modes of operations: Training and Inference. It is generally known that the inference can be done using lower-precision MAC units, but the training requires higher-precision MAC units. The lower-precision MAC units consume less energy which may be desirable for low-power applications. We propose an iterative MAC model where the inference will be done using low-precision MAC in a single pass, and the training will be done with the same low-precision MAC using multiple passes (to achieve higher bit precision). The proposed model, during training, determines the number of iterations on the fly by checking the error magnitude. Experimental results, with LeNet-300-100 model implemented using the iterative MAC, show a satisfactory performance for digit classification.

Preface

- Chapter 2 of this thesis has been published as {Machupalli R, Hossain M, Mandal M, "Review of ASIC accelerators for deep neural network" Microprocessors and Microsystems, Volume 89, 2022}. I was responsible for reviewing, classifying, and evaluating the existing accelerators. Dr. Masum Hossain provided the insights and helped me review the hybrid technology-based implementation of ALU units. Dr. Masum Hossain and Prof. Mrinal Mandal were the supervisory authors involved with concept formation and manuscript composition. This Chapter reviews existing ASIC DNN accelerators and classifies them into four categories. Dataflow architectures are evaluated for different workloads.

This thesis is dedicated to the memory of my father

MADDILET MACHUPALLI

Acknowledgment

I would like to express my sincere gratitude to my supervisors, Prof. Mrinal Mandal and Prof. Masum Hossain, for their continuous support, encouragement, and guidance throughout my graduate studies. All my achievements have benefited from their knowledge and inspiration. It would not have been possible to write this thesis without their help.

Special thanks to the defense committee chair Prof. Ying Tsui and committee members, Prof. Bruce Cockburn and Prof. Jie Han, for the valuable suggestions on the thesis.

Much respect to my labmates, classmates, and friends: Dr. Salah AL-Heejawi, Dr. Lina Liu, Ms. Shyama Gandhi, Mr. Naga Venugopal Kasibhotla, Mr. Raviraj Polishwala, Mr. Khimji Chavda, Mr. Bharath Tedlapu, Mr. Bhargav Tedlapu, Mr. Chetan Kumar and many others. They made my days meaningful and happy. The years I have spent at the University of Alberta have been an unforgettable period of learning and working experience.

My heartiest love and thanks to my parents and family members. Thank you for your tremendous love, devotion, and encouragement throughout my life, for which mere words cannot express my gratitude.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

ADC	Analog-to-Digital Converter
AI	Artificial Intelligence
AMI	Arithmetic Intensity
ASIC	Application-Specific Integrated Circuit
ALU	Arithmetic Logical Unit
BW	Bandwidth
CMOS	Complementary Metal–Oxide–Semiconductor
CNN	Convolutional Neural Network
CONV	Convolution
CPU	Central Processing Unit
CSC	Compressed Sparse Column
CSR	Compressed Sparse Row
DAC	Digital-to-Analog Converter
DFP	Dynamic Fixed-Point
DNNs	Deep Neural Networks
DNPU	Deep Neural Processing Unit
DRAM	Dynamic Random-Access Memory
DVAFS	Dynamic Voltage, Accuracy, and Frequency Scaling
FeFET	Ferroelectric Field-Effect Transistor
EIE	Efficient Interface Engine
FDSOI	Fully Depleted Silicon-On-Insulator

FFNs	Feed-Forward Networks
FIFO	First-In-First-Out
FIX	Fixed-Point
FL	Floating-Point
FPGA	Field-Programmable Gate Array
GPPs	General-Purpose Processors
GPU	Graphics Processing Unit
HMC	Hybrid Memory Cube
ILM	Improved Logarithmic Multiplier
INT	Integer
MAC	Multiplier and Accumulation Circuit
MCM	Multi-Chip Module
ML	Machine Learning
MLP	Multi-Layer Perceptron
MNIST	Modified National Institute of Standards and Technology
MSE	Mean Squared Error
NFU	Neuron Flow Unit
NLR	No Local Reuse
NN	Neural Network
NPU	Neural Processing Unit
OS	Output Stationary
PCM	Phase Change Memory
PEs	Processing Elements

PIM	Process-In-Memory
ReLU	Rectified Linear Unit
ReRAM	Resistive RAM
RF	Register Files
RNNs	Recurrent Neural Networks
RNS	Residual Number System
RS	Row Stationary
SCNN	Sparse Convolutional Neural Network
SIMD	Single-Instruction Multiple-Data
SNAP	Sparse Neural Acceleration Processor
STTMRAM	Spin-Transfer Torque Magnetic Random-Access Memory
SWP	Sub-Word Parallel processing
TDMS	Time-Division Mixed Signal
TOPS	Terra Operations Per Second
TPU	Tensor Processing Unit
UNPU	Unified Neural Processing Unit
WS	Weight Stationary

Chapter 1

Introduction

Artificial intelligence (AI) is the science and engineering of creating intelligent machines that have the ability to achieve goals like humans do [1]. Intelligence is the general mental ability for learning, reasoning, and problem-solving. The human brain consists of billions of neurons connected in a complex structure. Therefore, creating an intelligent model requires a large number of well-connected computing units (small building blocks) and enough examples to train the model. Due to the availability of a large quantity of data and computing resources in recent times, the creation of large-size AI models is realizable.

Machine learning (ML) is a subsection of artificial intelligence in which a mathematical model is trained over numerous examples to solve a new problem. ML is a rapidly evolving field in artificial intelligence due to the availability of a large set of example data for training. Deep Neural Networks (DNNs) are ML algorithms that use multiple neuronal layers to extract high-level features to classify or segment the input data. DNNs have been successfully applied to many problems, such as computer vision [2], robotics [3], security [4], medical diagnosis [5], self-driving [6], and natural language processing [7]. The DNN model size (e.g., number of layers, parameters) typically increases with problem complexity. Depending on the problem's complexity, a large amount of computing resources is typically required to implement a model. Hence, deploying DNN models on edge devices, where data is collected and processed near the sensor, is still limited.

The deployment of DNN models imposes severe design and scalability challenges on conventional embedded systems as they require substantial computing resources. General-purpose processors, like CPUs, have limited computing resources and fail to provide desired performance [96]. As an alternative, DNN models can be built on cloud computing servers. Cloud computing requires high bandwidth internet service to send and receive the data. Some applications require data processing at the edge devices for low latency, for example, in applications such as auto-piloted cars [6]. Therefore, the demand for ML/DNN accelerators has

been increasing recently [9,10,11,96]. Domain-specific accelerators provide better latency and higher energy efficiency [12, 13, 14, 15,16].

1.1 Motivation

The computational resources can be embodied on edge devices by integrating co-processors or accelerators like a Graphics Processing Unit (GPU), a Neural Processing Unit (NPU), and a Tensor Processing Unit (TPU) with the central processor. The co-processor/accelerators are optimized for high throughput, low latency, and low power for a specific category of applications. There are applications, like drone technology and mobile devices, where size and energy are more important because they run on batteries. The deployment on battery-powered edge devices needs an energy-efficient accelerator for longer battery life. Therefore, it is vital to study the existing accelerator architecture and identify potential areas to improve its flexibility, scalability, and power efficiency. An optimized accelerator should have the flexibility to process dense to highly sparse networks, low precision to high-precision dataflows at lower power consumption.

Many researchers from academics and companies like IBM, NVIDIA, AMD, and Google are working to develop specialized processor architecture or special hardware for DNNs. TrueNorth [8] from IBM, NVDLA [9] architecture from NVIDIA, Deep Learning Processor Unit (DPU) IP [10] from AMD-Xilinx, and TPU [11] from Google (deployed different versions in its servers and mobile platforms) shows the importance of DNN accelerators. Other well-known architectures from academic are Cnvltin [12, 13], Cambricon-X [14], Eyersis [15], Envision [16], EIE [17], SCNN [18], SIMBA [19], and many others. But can we use the existing accelerators in embedded systems? Are these accelerators flexible enough to provide optimal performance on all types of DNN models? Some accelerators are designed for sparse DNNs, and some are for dense DNNs [10, 11, 12, 16, 17]. Many of them are intended for inference. If an application requires fine-tuning of parameters at the edge device to mitigate environmental changes, then the accelerator should be capable of training the model. An accelerator designed for training a model is likely to underperform in energy efficiency for inference task due to performing low-precision operations using high-precision Arithmetic and Logical Units (ALU). For this reason, Google has different versions of TPU for training (server TPUs) and inference (edge TPUs). Therefore, it

is essential to understand how the existing accelerators work and identify the areas to incorporate flexibility.

The parameters such as energy, compute capacity, silicon area, and accuracy are interrelated in designing a processor architecture. Accuracy mainly depends on the precision and dynamic range of parameters. Precision and dynamic range can improve by the number of bits representing the parameters. The complexity in the datapath and memory bandwidth requirement will increase with the bit length (for parameter representation). The numerical format required for an ML is extensively investigated [20, 21, 22]. The inference requires lower precision compared to training. An architecture with reconfigurability in its precision can be used for training and inference. The work presented in this thesis studies the existing architectures and identifies areas to improve flexibility.

1.2 Neural Networks

Neural networks (NN) learn the decision ability based on experience, i.e., training. The NNs are composed of artificial neurons, also known as *perceptrons,* and somewhat mimic the human brain. A perceptron takes multiple inputs and produces a single output, as shown in Fig.1.1(a). Inputs are multiplied by synaptic weights (weight parameters), which refer to the strength or amplitude of the connection between two neurons (nodes). Fig. 1.1 shows the schematic of the simple and modern perceptron. Note that Fig. 1.1(b) has the advantage of incorporating nonlinearity and bias and is typically used in all current NNs and DNNs.

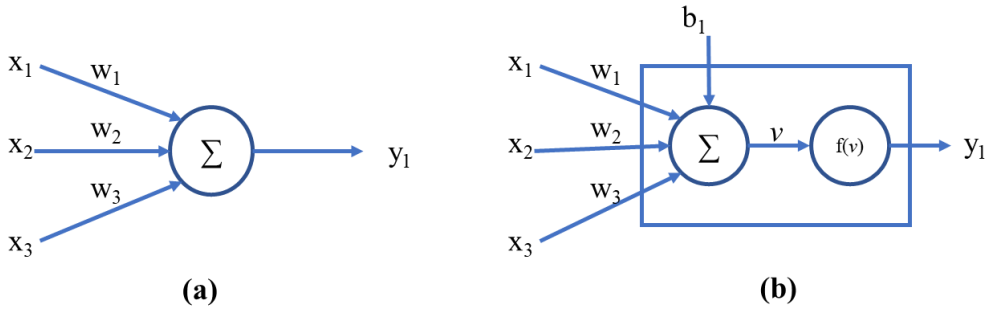


Figure 1.1. Basic neuron structure, (a) perceptron, (b) modern perceptron (artificial neuron) with non-linear function.

ML has various architectures such as Multi-layer Perceptron (MLP)/Feed Forward Networks (FFNs), Convolutional Neural Networks (CNNs), Deep Neural Networks (DNNs), and

Recurrent Neural Networks (RNNs). Different types of neural networks are better suited for different kinds of tasks and input data.

Multi-layer Perceptron:

A multi-layer perceptron (MLP) is a type of neural network that consists of a series of fully-connected layers. The output of each neuron in a layer is connected to the input of every neuron in the next layer. MLPs are often used for tasks such as classification and regression.

Convolutional Neural Network:

A convolutional neural network (CNN) is designed to process grid-like data, such as images. CNNs consist of a series of convolutional layers, pooling layers, and fully-connected layers [2, 95]. Convolutional layers apply filters to local regions of the input data to extract features while pooling layers down-sample the feature maps to reduce the computational cost of the network. In CNN, each neuron will act as a filter, convolving with input data to extract features. CNNs are commonly used for image classification and object detection.

Deep Neural Network:

A deep neural network (DNN) is a neural network with many layers. The term "deep" refers to the large number of layers in the network. DNNs can be seen as an extension of CNNs to handle more complex input data and perform better in a broader range of tasks.

Recurrent Neural Network:

A recurrent neural network (RNN) is designed to handle sequential data, such as time series data or natural language. The RNNs consist of a series of recurrent layers, where each neuron receives input from the previous layer and the previous time step. RNNs capture information from the entire sequence of inputs rather than just the current input. RNNs are often used for tasks such as speech recognition and language translation.

The working principle behind all the models is similar. For a classification task, key features are generated from the data, and the classification is done based on these features. It can also be viewed from a different angle: the lower dimensional input data is projected into higher dimensional feature space, and a separation curve is learned for classification. The new features

are application and input dependent. Therefore, different applications may require different NNs (i.e., different architectures or weights). The NNs consist of weight parameters that generate higher dimensional values or features from the input data. There can be weights with zero value, which means that the input feature through that weight has no importance in the neuron output.

The NNs have two stages: predicting the output based on the input values is called a forward pass or inference, and learning the optimal weights is called training. As the definition of machine learning, it learns decision ability based on experience. In NN, experience means training in which a possible range of input values and their outputs is provided. First, the inputs are processed through the network to generate outcomes. The generated output is then compared with the desired result, and any errors in the output are propagated back through the network to update the weights. Consider an MLP network for mathematical modeling, which can easily be extended to all other NNs. To generalize deeper networks, we follow specific notation and follow the same in the rest of the document. The layer number is denoted with l on the superscript of parameter, j indicates neuron number in the l^{th} layer, and i indicates neuron in $(l\text{-}1)^{th}$ layer. The weight matrix is denoted with W, and individual weights are represented with w_{ji}, the weight between the j^{th} neuron in the l^{th} layer and i^{th} neuron in $(l\text{-}1)^{th}$ layer.

1.2.1 Inference/ Forward Pass

A neural network can be represented mathematically as a function that takes the inputs, X, and predict the outputs, Y. Each neuron in the network can be described as a node in a computational graph. Fig. 1.1(b) shows the schematic of a neuron whose output y is calculated as follows.

$$v = W \times X + b$$

$$y = f(v) = f(W \times X + b) \tag{1-1}$$

where X is the input vector, W is the weight matrix, b is the bias, v is the weighted sum, and f is the activation function. Similarly, the l^{th} layer in an MLP network shown in Fig.1.2 can be written as

$$Y^l = f(W^l \times X^l + b^l) \tag{1-2}$$

where Y^l, W^l and b^l are l^{th} layer output, weights, and biases, respectively. Note that the input to the l^{th} layer, X^l, is the output of the $(l\text{-}1)^{\text{th}}$ layer, i.e. $X^l = Y^{l-1}$.

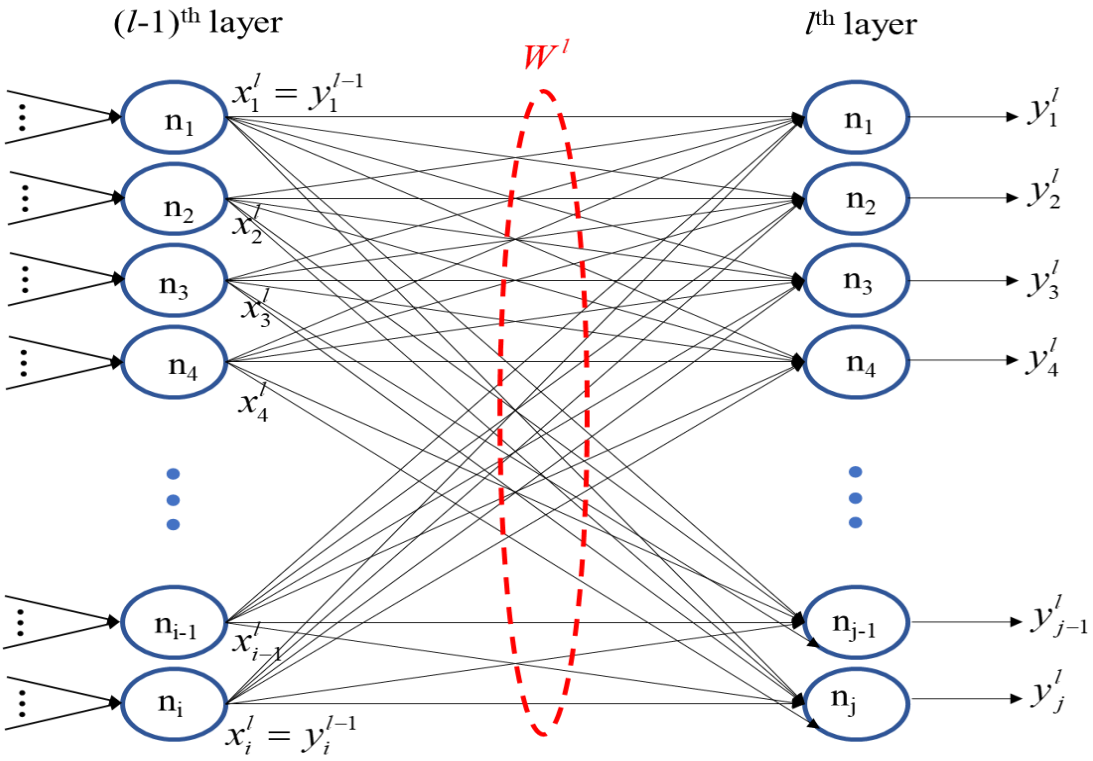


Figure 1.2. Data flow graph in an MLP network with each neuron equivalent to Fig. 1.1(b).

The activation function, f, is a non-linear function that determines a neuron's output. Common choices for f include the sigmoid function, the hyperbolic tangent function, and the rectified linear unit (ReLU) function.

1.2.2 Backpropagation

Once the input data is processed through the forward pass, an error (e) and a cost function (C) are defined to measure the difference between predicted outputs and desired outputs (targets). Let Y denote the targets. If the network has L layers, then the L^{th} layer outputs, i.e., Y^L are the predicted outputs. Error (e) is defined as the difference between the target and predicted values as follows:

$$e_j = y_j - y_j^L \tag{1-3}$$

where j indicates a neuron number in the l^{th} layer.

The cost function (the loss function) is a mathematical measure that quantifies how well the network performs on a given task. The cost function compares the network's predicted output to the desired output and computes a scalar value representing the error between the two. The goal of training a neural network is to minimize the cost function, reducing the error and improving the network's performance. Different types of tasks and networks may require different cost functions. Some cost functions are Mean Squared Error (MSE), Binary Cross-Entropy, Categorical Cross-Entropy, and Kullback-Leibler Divergence (KL-Divergence).

Mean Squared Error (MSE) is a commonly used cost function for regression tasks. It calculates the root mean squared difference between the predicted and true values as follows:

$$C = \frac{1}{2}\sum_{j}\left\|e_j\right\|^2 = \frac{1}{2}\sum_{j}\left\|y_j - y_j^L\right\|^2 \tag{1-4}$$

where L represents the number of layers in a network and y_j^L represents the output of j^{th} neuron in the last (L^{th}) layer. Binary Cross-Entropy is used for binary classification tasks. It measures the difference between the predicted and the actual probabilities of the positive class. Categorical cross-entropy measures the difference between the predicted and actual probability distributions, used for multi-class classification tasks. Kullback-Leibler Divergence (KL-Divergence) is another commonly used cost function for multi-class classification tasks. It measures the difference between the predicted and true probability distributions but with a different mathematical formulation than categorical cross-entropy.

It is important to note that the choice of the cost function can significantly impact the neural network's performance. Using the wrong cost function can lead to suboptimal results or prevent the network from learning. In addition, some cost functions may have limitations on the types of problems they can handle, the types of networks they can use, or the range of values they can generate. The specific task, numerical stability, and network architecture typically guide the choice of the cost function. It may also be helpful to experiment with different cost functions during training to see which one works best for a given problem.

The most important job while training a NN is calculating the weight gradients. The weight gradients are the partial derivatives of the cost function with respect to the weights of a neural network. During the training process, the goal is to find weights that minimize the cost

function. Training is typically done using a gradient descent algorithm [97]. The weight gradients tell us how much the cost function changes for a slight change in each weight. The gradients for lower layers are calculated using the chain rule. Hence, first, calculate the gradient with respect to the last layer weights w_{ji}^{L}. We consider the MSE cost function defined in Eq. 1.4 for the rest of the derivatives, the same can be extended for any cost function. Using Eq. 1-1, 1-2, 1-3, and 1-4, the weight gradients of the last layer can be written as follows.

$$\frac{\partial C}{\partial w_{ji}^{L}} = \frac{\partial C}{\partial e_{j}^{L}} \frac{\partial e_{j}^{L}}{\partial w_{ji}^{L}(n)} \tag{1-5}$$

Using Eq. 1-4, we obtain:

$$\frac{\partial C(n)}{\partial e_{j}^{L}(n)} = e_{j}(n).$$

From Eq. 1-3, we obtain:

$$\frac{\partial e_{j}^{L}}{\partial w_{ji}^{L}} = 0 - \frac{\partial y_{j}^{L}}{\partial w_{ji}^{L}} = -\frac{\partial y_{j}^{L}}{\partial w_{ji}^{L}}.$$

Substituting Eq. 1-2 in the above equation, we obtain:

$$\frac{\partial e_{j}^{L}}{\partial w_{ji}^{L}(n)} = -f'(.) \times x_{i}^{L}$$

where $f'(.)$ is the derivative of the activation function (f) with respect to the weighted sum (v). Substituting the above in Eq. 1-5, we obtain:

$$\frac{\partial C}{\partial w_{ji}^{L}} = -e_{j}(n) f'(.) x_{i}^{L} \tag{1-6}$$

In the backpropagation algorithm, the weight correction term Δw_{ji} (to be applied to w_{ji}) is defined as

$$\Delta w_{ji} = -\eta \frac{\partial C}{\partial w_{ji}} \tag{1-7}$$

where η is the *learning-rate* parameter. The *-ve* sign accounts for the gradient descent in weight space. Substituting Eq. 1-6 in Eq. 1-7 yields:

$$\Delta w_{ji}^{L} = \eta \times e_j \times f'(.) \times x_i^{L} \tag{1-8}$$

Generalizing Eq. 1-8 to all the layers, a *local gradient* δ can be defined as

$$\begin{aligned}\delta_j &= -\frac{\partial C}{\partial v_j} \\ &= -\frac{\partial C}{\partial e_j} \times \frac{\partial e_j}{\partial y_j} \times \frac{\partial y_j}{\partial v_j} \\ &= -e_j \times (-1) \times f'(v_j) \\ &= e_j \times f'(v_j)\end{aligned}$$

where v_j is called a local response, which is the input to the activation function. In the l^{th} layer, v_j can be expressed as $v_j^l = \sum_i w_{ji}^l x_i^l + b_j^l$. The weight gradients can be written in terms of local gradients as follows:

$$\Delta w_{ji}^{L} = \eta \times \delta_j^{L} \times x_i^{L} \tag{1-9}$$

The backpropagation algorithm passes local gradient values back to lower layers to calculate the weight gradients. The data flow in the backpropagation is shown in Fig. 1.3. Figure 1.3(a) shows the backpropagation data flow in a single neuron. Fig. 1-3(b) shows data flow in the l^{th} layer of an MLP network. From Eq. 1-2,1-6, and using the chain rule in partial derivatives, the local gradients of the lower layer can be calculated as follows:

$$\delta^l = [W^{l+1}\delta^{l+1}] \times f'(.)$$

The weights are updated by adding weight gradients at each iteration. The updated weights can be written as follows:

$$w_{ji}^l(n+1) = w_{ji}^l(n) + \Delta w_{ji}^l(n)$$

where *n* represents the iteration number.

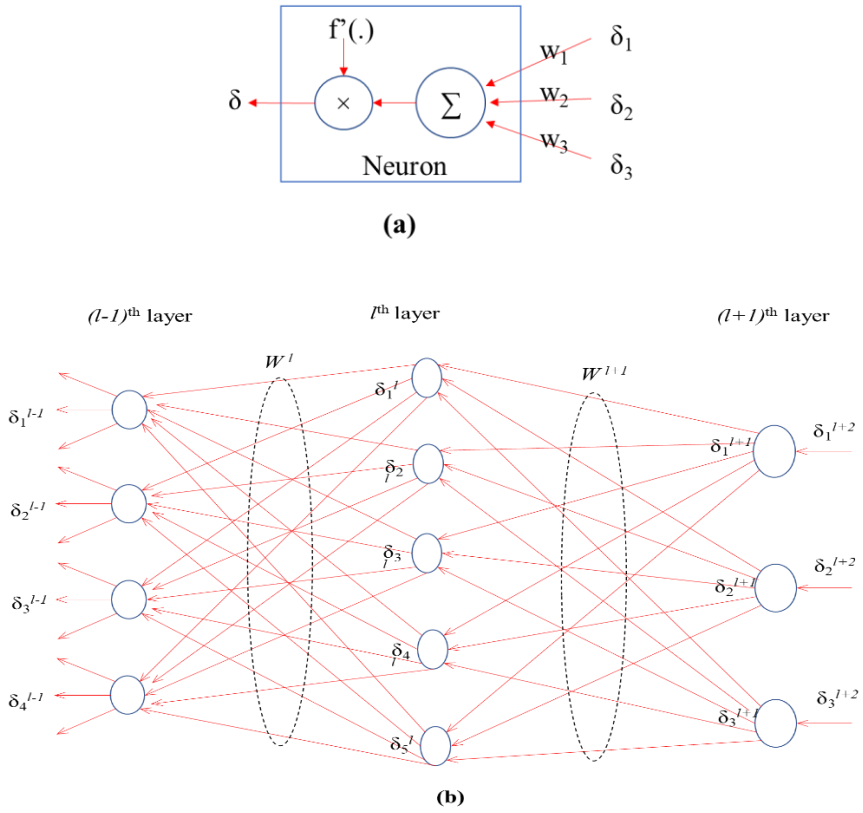


Figure 1.3. Data flow graph of neural network in backpropagation. (a) Data flow in a single neuron, (b) Dataflow at the l^{th} layer of an MLP network with each neuron equivalent to (a).

1.3 Contributions and Thesis Outline

The rest of the thesis is organized as follows: Chapter 2 reviews the state-of-the-art ASIC DNN accelerators. In the existing architectures, we identify three key areas, ALU, dataflow, and sparsity, which can potentially improve the overall performance of an accelerator. Existing accelerators for inference are broadly classified into four categories. Each area offers multiple tuning techniques to improve the overall architecture performance. The advantages and drawbacks of each category are discussed. Chapter 3 proposes an iterative MAC unit, where higher-precision arithmetic calculation can be performed iteratively using a lower-precision MAC unit. This MAC unit can be used to implement a DNN inference model (with a single iteration) and can be used to train the DNN model (with multiple iterations), which requires higher-precision calculations. The effectiveness of the proposed iterative MAC was evaluated by simulating the LeNet-300-100 model. Finally, the conclusions and future research directions are presented in Chapter 4.

Chapter 2

Review of ASIC Accelerators for Deep Neural Network

Deep neural networks (DNNs) have become an essential tool in artificial intelligence, with a wide range of applications such as computer vision [2], medical diagnosis [5], security [4], robotics [3], and autonomous vehicles [6]. The DNNs deliver state-of-the-art performance in many applications. The complexity of the DNN models generally increases with application complexity, and deploying complex DNN models requires high computational power. General-purpose processors are unable to process complex DNNs within the required throughput, latency, and power budget [96]. Therefore, domain-specific hardware accelerators are required to provide high computational resources with superior energy efficiency and throughput within a small chip area. In this Chapter, existing DNN hardware accelerators are reviewed and classified based on the optimization techniques used in their implementations. Each optimization technique generally improves one or more specific performance parameter(s). For example, the hardware optimized for sparse DNNs may provide poor performance for dense DNNs in terms of power and throughput. Therefore, understanding the tradeoff between different hardware accelerators helps to identify the best accelerator model for application deployment. We identify four major areas, ALU, dataflow, sparsity, and Hybrid model in hardware architectures that can potentially improve an accelerator's overall performance. Existing hardware accelerators for inference are broadly classified into these four categories. It is difficult to compare the existing accelerators based on speed and energy as each accelerator has its own specifications such as number of MAC units, on-chip memory size, sparsity in data, and the DNN model. For example, it is difficult to compare the EYERISS [15] accelerator (optimized for sparse data and row-stationary dataflow) and the TPU accelerator [11] (optimized for matrix multiplication and weight stationary dataflow). The classification model can help to identify appropriate performance parameters and benchmarks for accelerators.

2.1 Introduction

Artificial intelligence is the ability of a system to think, learn, and react like humans without explicit programming. The human brain consists of billions of neurons connected in a complex structure with operational efficiency. Similarly, the creation of an intelligent model requires a large number of well-connected computing units (or small building blocks) and enough examples to train the model. Due to the availability of a large quantity of data and computing resources in recent times, the creation of intelligent machines is realizable. Machine learning (ML) is a subsection of artificial intelligence in which a mathematical model is trained over many examples to solve a new problem. Deep neural networks (DNNs) are subsections of ML with a deep network structure and shared weights (filters).

The DNNs have been successfully applied to many problems, such as computer vision [2], robotics [3], security [4], medical diagnosis [5], and self-driving cars [6]. Most DNNs are based on the convolutional neural networks (CNN), where output feature maps are typically generated by convolving input feature maps with 3D filters. Recent DNN models have been shown to surpass human performance in some applications. The performance improvements typically come with the increased complexity of the DNNs. As seen in Table 2.1, the classification of a small-size image (e.g., 227×227 pixels) requires billions of arithmetic operations (i.e., multiplication and addition). The MCN-MobileNet has 4.19 million parameters (weights) and requires 0.58 billion operations to classify an image. A large-size VGG-19 DNN model requires about 20 billion operations per classification.

Table 2.1. Number of parameters and operations required for different DNN models [23].

Model	**Input size**	**PARAMETERS SIZE** (In millions)	**# of Operations** (In GOPs)
AlexNet	227 x 227	61.07	0.73
Squeezenet	224 x 224	1.31	0.84
VGG-16	224 x 224	138.41	16
VGG-19	224 x 224	143.65	20
GoogleNet	224 x 224	13.36	2
Resnet-18	224 x 224	11.79	2
Resnet-152	224 x 224	60.29	11
Inception-V3	299 x 299	23.85	6
Densenet-201	224 x 224	20.18	4
MCN-mobileNet	224 x 224	4.19	0.58

General-purpose processors, like CPUs, are unable to provide such huge computing power with the required latency. To deploy the DNNs in real-time applications, the embedded processors must have high throughput and low power consumption. Therefore, the demand for domain-specific accelerators has been increasing in recent times as these accelerators can provide superior performance at higher energy efficiency.

There are two main phases in DNNs: training and prediction (or inference). In the training mode, an example input with a known outcome is applied to the model to learn its internal parameters. In the prediction (or inference) mode, the possible outcome is calculated based on the input test data. Training typically requires high-precision numerical representation, while low-precision representation is enough for inference [20, 21, 24, 25]. Generally, training is done using high-power GPUs and data centers. The precision and size of the trained DNN models can be reduced significantly with negligible (<1%) change in the accuracy for inference [22]. In real-time deployment, the trained DNNs need to operate in inference mode only if there is no change in application requirements. Therefore, hardware accelerators for inference mode are more important than for training. Therefore, this paper is mainly focused on the inference mode, and all discussions are subject to the inference mode of operation.

Globally, a large number of researchers, in both academia and industry, are working towards developing optimized hardware for DNNs inference. DNN accelerators have been developed using FPGAs, GPUs, and ASICs. GPUs come with massive parallel compute units and process DNN computations in parallel. GPUs are power-hungry, which limits their applications in embedded and battery-powered systems. FPGAs have high performance per watt and can be configurable in the fields. FPGAs are often used to prototype and validate the design. ASICs are custom-designed for specific applications with optimum speed and power consumption. ASICs are best suited for embedded devices. ASIC implementation takes longer development cycle compared to GPUs and FPGAs and have no flexibility after design. Talib et al. [26] reviewed several hardware accelerators for machine learning using FPGA, GPU, and ASIC platforms and discussed the advantages of each platform over other platforms. Guo et al. [27] surveyed several FPGA-based neural network accelerator designs and summarized the methods used for design automation. The FPGA allows less control and flexibility over the multiplication and accumulation (MAC) unit design, which typically limits the exploration of the

MAC variants. Li et al. [28] presented an overview of the GPU, FPGA, and ASIC-based accelerators and a detailed explanation of the DianNao [29] family of accelerators. This has motivated significant progress in the ASIC accelerators after the [28] survey. Camus et al. [30] analyzed the precision-scalable MAC units from different accelerators and discussed their benefits in different scenarios. Although the MAC unit is an important block in the DNN accelerator design to improve performance, the MAC alone cannot define the overall performance. Hence, along with the MAC unit, some other factors in the accelerators need to be analyzed. The MAC utilization depends on the data flow and on-chip memory. An efficient architecture should have high MAC utilization, i.e., MAC should not be idle because of operands are unavailable. Reuther et al. [31] discussed existing ML accelerators based on peak performance vs. power scatter plots. The accelerators are broadly categorized into six types based on regions in the plot. The factors causing variation in the performance of different accelerators are not well explained in [31].

Du et al. [32] presented an overview of self-aware neural network systems, where a system can predict and adapt dynamics in network parameters, such as precision, sparsity, and network structure, based on the input data. The self-aware techniques can significantly improve the accelerator's throughput and energy efficiency, but the accelerators should have some flexibility. For example, a DNN with a variable precision requirement at different layers needs a variable precision MAC to adapt and save energy. The survey did not include much information on the implementation techniques to incorporate the flexibility in accelerator implementation and how it affects overall performance. Sze et al. [24] provided an overview of the DNN development platforms, optimization algorithms, accelerator implementations, and benchmarks. The paper explains three different dataflow methods but does not include all recent advances in the arithmetic logic unit and sparsity exploration. Chen et al. [33] reviewed several current DNN accelerators based on their application and technologies used (e.g., ReRAM, Hybrid Memory Cube). Most surveys provide the architectural and performance improvements of existing DNN accelerators, but analyzing the architectures in a generalized framework would be helpful.

The existing literature classifies the different DNN accelerators based on their implementation techniques or applications. For example, the accelerators in [33] are reviewed based on architectures (e.g., stand-alone or co-processor-based) or technologies used (e.g., Re-

RAM, HMC). Similarly, the accelerators in [31] are classified based on the peak power versus performance tradeoff. In the DNN literature, we identify three major areas for improvements in the DNN architecture: Arithmetic logic unit, dataflow, and sparsity. In this Chapter, we present a comprehensive review of the ASIC accelerators for the DNN architectures. The state-of-the-art accelerators are classified into three broad categories (i.e., ALU, data flow, and sparsity-based) based on their architectural differences. This broad classification can provide more insights to develop generic DNN architectures. Additionally, we have added a fourth section that captures a recent trend of analog-digital hybrid digital implementation for faster computation.

The organization of the Chapter is as follows. Section 2.2 presents the background information and performance criteria of hardware architectures. Section 2.3 presents a comprehensive review of the DNN hardware architectures and their classification. Section 2.4 presents evaluation methods and observations in existing accelerators, followed by the conclusions in Section 2.5.

2.2 Background

The superior performance of DNNs generally comes at the cost of high computations. For example, AlexNet [2], which won the ImageNet challenge [34] in 2012, has 61M parameters and requires 727M MAC operations per image classification. Large DNNs may require billions of MAC operations per inference, as shown in Table 2-1. Performing such a large number of operations sequentially affects the throughput. Existing general-purpose processors (GPPs) may be unable to provide the required computational power and throughput within a low-power budget. The GPU can provide high computational speed but consumes a large amount of power. GPUs can therefore be used on servers where the computational speed is more important than the power requirement. Domain-specific accelerators (e.g., ASICs) are known to provide high energy efficiency (around 1~10 TOPs/W). The FPGAs have less energy efficiency but have the advantage of reconfigurability.

Real-time deployment of DNN is constrained by energy efficiency and the throughput of embedded processors to maximize battery life. For example, a typical mobile phone has a 2-3 Ah (5V) battery life (i.e., 15 Wh) and the DNN processing power should be only a fraction of the maximum available power. For real-time data processing, the processor should have a

throughput equal to the data collection frame rate (e.g., the camera frame rate). Fortunately, there is no interdependency among outputs (i.e., in the same layer output or feature map) in a DNN layer. Therefore, parallel implementation of large MAC units can increase the throughput. An example of the DNN accelerator implementation based on Parallel MAC units is shown in Fig. 2.1. In general, the size of the accelerator in silicon and the power requirements are directly proportional to the number of MAC units (working in parallel) and on-chip memory. Note that the DNN-specific accelerators will have an array of processing elements (PEs) connected to their neighbors. Each PE contains one to several MAC units connected in such a way that matrix multiplication can be performed with a single instruction. The MAC unit contains a control unit to configure the operation to multiplication or addition or both and register files to store the local parameters and intermediate results. To increase the on-chip storage, global and local buffer memory blocks are implemented along with PEs.

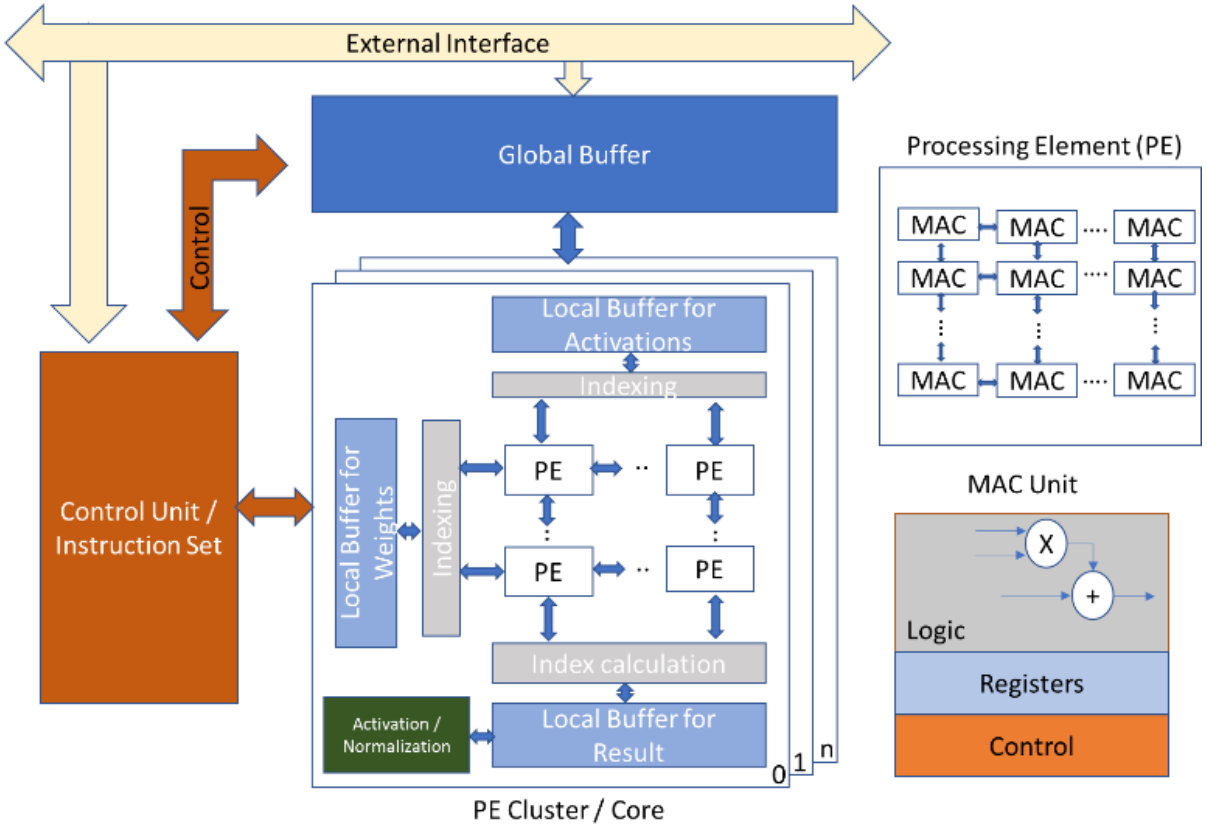


Figure 2.1. Block diagram of a generic DNN architecture.

The cost of the individual MAC units can be reduced with lower bit-length/precision of the MAC units. The energy and area consumption of multiply and add circuits for four different

precisions are shown in Table 2.2. An 8-bit fixed-point (FIX) add circuit occupies 116× less area and consumes 30× less energy (in picojoules) than a 32-bit floating-point (FL) adder. For multiplication, an 8-bit fixed-point circuit consumes 18.5x less energy and occupies 27.3x less area than a 32-bit floating-point. Approximately, the energy and area of fixed-point circuits scale linearly for add, quadratically for multiply, with the number of bits [35].

Reduction in the MAC precision can save both the computation and storage requirements. Therefore, the impact of low precision on the accuracy of DNN models has been explored in the literature, mainly with respect to quantization [20, 22, 37]. In most DNNs, quantization of weights and activations to less than 16-bit integers can still provide accuracy similar to that of a 32-bit floating-point [37, 38, 39]. Linear quantization to 8-bit fixed-point numbers benefits the hardware implementation of the MAC unit, as shown in Table 2.2. Both energy consumption and silicon footprint increase with the increase in precision when changed from fixed-point to floating-point representation. In Binary network [40], weights and activations are quantized to binary values +1 or -1. The binarization of the network will simplify the multiplication into the XOR operation. Ternary network [41] quantizes the parameters to three levels: -1, 0, and +1. But applications of Binary and Ternary networks are limited.

Table 2.2. Resource consumption of MAC units at different precisions [36]

Operation/ Precision	**Energy (*pJ*)**		**Area (μm^2)**	
	MUL	**ADD**	**MUL**	**ADD**
8-bit fixed	0.2	0.03	282	36
32-bit fixed	3	0.1	3497	137
16-bit float	1.1	0.4	1640	1360
32-bit float	3.7	0.90	7700	4184

Depending on the application requirements, the arithmetic operations in a DNN network may be implemented using different bit precisions. Also, there exist models whose optimized bit length varies from layer to layer. For example, for a 5-layer Convnet (with three convolutional and two fully-connected layers), the optimized bit length requirement for the five layers has been found to be 8-7-7-5-5 bits [42]. In other words, no standard precision requirement is optimized

for all layers or models. Therefore, a flexible DNN hardware accelerator (or the associated MAC units) should be able to support all possible bit precisions. For lower-precision computations, multiple operations can be performed with a single MAC unit by hardware reuse or sub-word parallel processing. With hardware reuse, the overall throughput or peak performance of an accelerator can be improved for lower-precision layers or models. For example, the Tesla T4 [43] GPU can be configured to four precisions: 4-bit, 8-bit, FP16/FP32-mixed, and FP32. Tesla T4 achieved the highest speed at thelowest precision (4-bit). The throughput increases at the cost of reduced precision. The peak performance is typically expressed in arithmetic operations per sec (OPS), primarily depending on the available MAC units. An additional control unit is required to configure the MAC unit into multiple sub-MACs or bit length in a variable precision MAC unit. The overall size of the MAC unit increases with flexibility (in precision). In other words, the MAC density (i.e., MAC units per unit area) decreases with increased flexibility [30]. Therefore, there is a tradeoff between MAC's flexibility and density.

Having a large array of MAC units with a variable bit precision can fulfill the DNN processing requirement in terms of computations. But just having an extensive array of MAC units does not improve the throughput. To provide operands to all MAC units in a large array, sufficiently high memory bandwidth (BW) is required. After a certain point of arithmetic intensity, the memory bandwidth of an accelerator will determine the overall throughput. Fig. 2.2 shows the estimated roofline model for DNN inference on four different embedded platforms. Arithmetic intensity (AMI), also commonly referred to as the operational intensity or compute-to-communication ratio, is expressed as the number of arithmetic operations performed per byte of off-chip memory traffic (expressed in operations/byte). The arithmetic performance of the hardware depends on the AMI and the data access rate from the external memory. In other words, the arithmetic performance can be expressed as follows:

$$Arithmetic\ Performance = \min(PP, AMI \times BW) \tag{2-1}$$

where BW is the memory bandwidth, and PP is the peak performance. It is observed in Fig. 2-2 that the arithmetic performance increases initially with an increase in the AMI until peak performance (PP) is reached. After achieving the PP, any further increase in the AMI does not increase performance. The arithmetic performance is observed to be memory-bound when the AMI is to the left of the break point and compute-bound when the AMI is to the right [44].

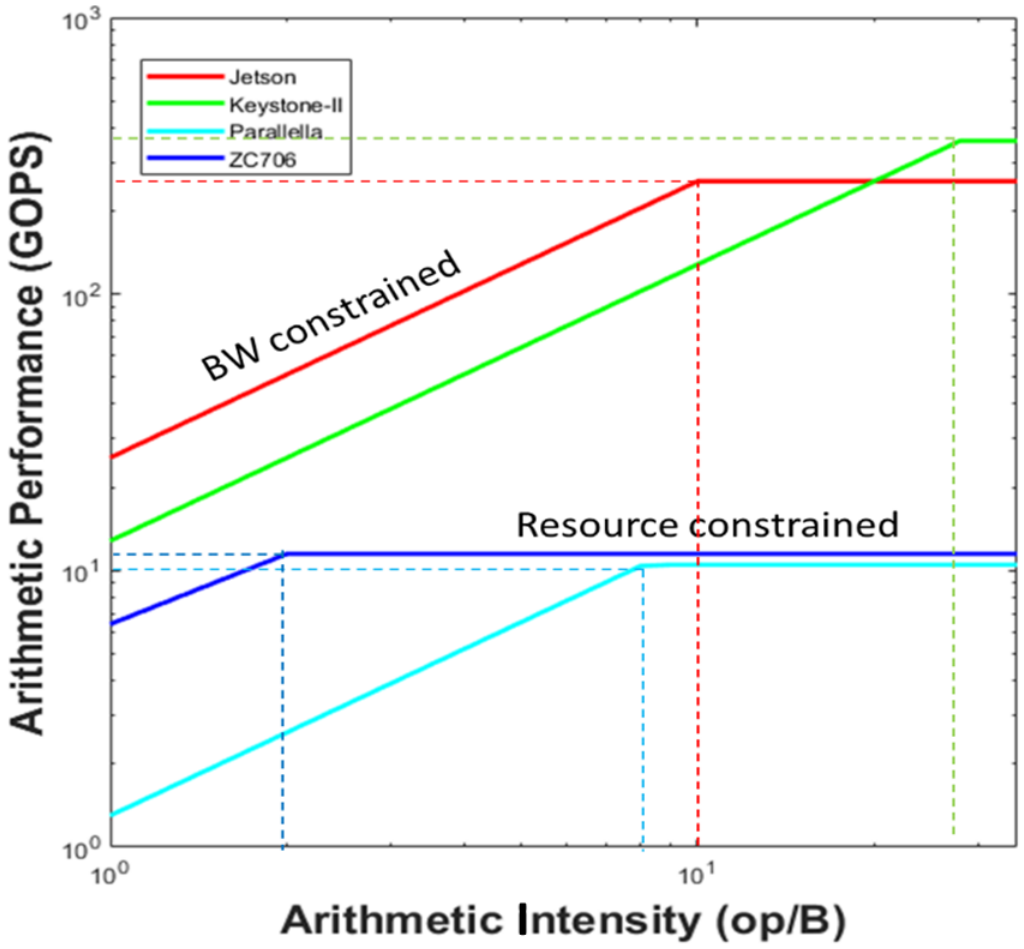


Figure 2.2. Comparison of the roofline models for DNN inference [44].

The arithmetic performance of the hardware typically depends on the PP, AMI, and memory bandwidth. Resources available on the chip define the PP of the hardware. Arithmetic intensity depends on the dataflow structure implemented and available on-chip memory. Note that an external memory operation is energy and time-consuming. Hence, the hardware should run at a minimum bandwidth to save energy. With minimum bandwidth, the arithmetic performance of hardware can be increased with increased AMI. As seen in Fig. 2.2, the arithmetic performance improves with an increase in the AMI in the linear region of the curves (as the PP and BW are constant). The AMI can vary through the data flow structure. Therefore, the dataflow structure should be optimized to achieve higher arithmetic performance for a given bandwidth.

Table 2.3. Memory hierarchy in a general accelerator and its approximate performances [24].

Memory level	Access time (approx. cycles)	Available capacity	Energy consumption (normalized)
Registers	1	< 0.5 KBs	1x (Reference)
PEs cache	2-4	~1-10 KBs	2x
Local buffer	10	~100 KBs	4x
Global buffer	40	~10 MBs	6x
Main memory	200	In GBs	200x

To avoid the data read/write each time (to speed up the computation, reduce energy consumption, and increase the AMI), the read data must be used as much as possible within the chip before writing it back to the memory. Fortunately, the convolution layers in DNNs have this data reuse option. For example, a single filter is reused to calculate all pixels in an output feature map. Therefore, reading the coefficients of a filter once is enough. But keeping all filter coefficients at each MAC unit is a resource (i.e., memory) consuming option. To reduce the overall energy cost of data movement, several levels of memory (e.g., global buffer, local buffer, registers) can be implemented in hardware. A rough estimation of the available memory size, latency, and energy consumption per operation at various levels are shown in Table 2.3 [24]. The global buffer (with a size of hundreds of kilobytes) connects to DRAM, with the local buffer dedicated to a few processing elements (PE). Read/write data from a Global buffer to a MAC consumes around six times more energy and 40x latency than read/write from register files. Register files (RF) corresponding to a MAC unit of a PE are connected to a local buffer and consume the least amount of energy to read/write the data. The advantage of the local buffer is limited by its available size. The energy consumption and access time increase from low-level memory (Registers) to high-level memory (Global buffer).

In a DNN, the output of a convolution or fully-connected layer goes through an activation function. The Rectified Linear Unit (ReLU) is a nonlinear activation function widely used in DNNs (all the networks in Table 2.1 uses ReLu) which maps the output value of a feature map as follows.

$$y = \begin{cases} x & x \geq 0 \\ 0 & x < 0 \end{cases} \tag{2-2}$$

where x is the input and y is the output of the activation function. It is observed that the negative output values are truncated to zero by the activation function. This truncation can make the output values sparse. It has been shown that the AlexNet has a sparsity between 19% to 63%, where the sparsity is defined as the percentage of the data (e.g., feature maps, filter coefficients) that are zero. Several researchers have exploited the sparsity in a DNN to increase the throughput and reduce power consumption.

The DNN model size (i.e., the number of the DNN weights) can be reduced through pruning without affecting the model accuracy. The pruning eliminates insignificant connections or weights (i.e., making the insignificant weights zero) in a DNN. Note that multiplication with a very small value operand results in a negligible value that will not likely alter the outcome. This observation makes the case for opportunistic energy savings by eliminating insignificant multiplications. The DNN architectures can therefore be designed to skip multiplications with zeros, known as zero skipping.

If arithmetic hardware can skip zero multiplication, sparsity in data and zero weights cumulatively reduce the computing power requirement. Higher speed can also be achieved by exploring sparsity in data. To exploit the sparsity further, the storage requirement can be reduced by encoding the sparse data. The compression techniques may vary from simple run-length coding to compressed sparse column (CSC) or compressed sparse row (CSR) [45]. Compression techniques, however need additional encode and decode modules in the hardware.

Based on the above discussion, it can be inferred that an efficient hardware accelerator must be optimized for low-precision, best data flow, and be flexible for varying precision and sparse models. As expected, there is a tradeoff between flexibility and optimized architecture. An architecture optimized for sparse models will affect the throughput of dense models. Accelerators optimized for the convolutional layer may not perform well on a fully-connected layer due to the data reusability. In a convolutional layer, weights are reused, but in fully-connected layers, input features are reusable for optimal performance. Overall, efficient

hardware for DNNs should have scalable precision to support different DNN models, optimized data flow structure to increase the arithmetic intensity, and should utilize sparsity.

The deployment of DNNs in real-time applications requires low-power and high-throughput DNN accelerators. Many efficient DNN accelerator architectures have been proposed over the last decade to reflect versatile efforts to improve the overall performance of the DNNs. Domain-specific accelerators will always have a scope to improve the overall performance by customizing architectures towards a specific application. Even the accuracy requirement of the same application can make a difference in the DNN complexity. A generalized DNN accelerator architecture should have the flexibility to work on different models at the optimum performance.

The DNN architectures can be broadly divided into three categories based on the area where the architecture has been primarily optimized. These three areas are Arithmetic logical unit (ALU), Dataflow, and Sparsity. In the ALU category, the basic building block, i.e., the MAC units (or an array of MAC units), are modified such that the accelerator can have large computing resources and flexibility to achieve the optimal performance with variable bit precision. In the Dataflow category, the parameters (e.g., weights, activations, partial sums) are managed such that the overall (intra-chip) data movement energy is reduced, and high arithmetic intensity (Ops/Byte) can be achieved. In the Sparsity category, the unstructured sparse data is managed such that the matrix multiplication units (e.g., a 2-D array of MAC units) can avoid the zero multiplications effectively. A comprehensive review of the DNN architectures based on these three criteria is presented in the following.

2.3 Classification

2.3.1 ALU-based Accelerators

Computation-hungry DNN algorithms require a huge amount of computing hardware resources. Large arrays of PEs are typically implemented in parallel to improve the computational power of a processor. Graphical Processor Units have thousands of PEs in parallel. Hence, GPUs are widely used as accelerators for DNNs. The GPUs can provide the throughput requirement but consume high energy. The energy consumption of a MAC unit can be reduced by decreasing the bit length. Therefore, low-precision DNN accelerator architectures have been proposed for DNN inference.

Chen et al. [46] proposed an architecture known as the DianNao architecture, with a Neuron Flow Unit (NFU) as the basic arithmetic building block. An NFU has 16 neurons, with each neuron having sixteen 16-bit fixed-point multipliers in stage 1 and 15 adders in a tree structure at stage 2 to add the multiplication results. Stage 3 has an activation layer. DianNao has three memory blocks, input buffer, output buffer, and synapse buffer, to store inputs, outputs, and weights, respectively. Based on the DianNao architecture, a series of accelerators DaDianNao [47], ShiDianNao [48], and PuDianNao [49] have been proposed by improving the NFU unit as well as dataflow. The DianNao family can provide 450x speedup and 150x reduction in energy with 64 chips over a GPU [29]. Although the Diannao family provides a good speed-up, it does not support variable precision. Running a four or 8-bit DNN Model will consume as much energy as the 16-bit model.

To save energy at lower precision, the Dynamic Voltage, Accuracy, and Frequency Scaling (DVAFS) MAC-basedd CNN architecture (ENVISION) has been proposed in [16]. In DVAFS, all run-time adaptable parameters influencing power consumption: activity (α), frequency (f), and voltage (V) are scalable. The dynamic power consumption at constant throughput is given by [16]

$$P_{DVAFS} = \frac{\alpha}{k_1} C \frac{f}{N} \left(\frac{\alpha}{k_2} \right)^2 \tag{2-3}$$

where k_1, k_2, and N are scaling factors of switching activity, voltage, and level of parallelism, respectively. For lower precision, the switching activity can be reduced by masking lower LSBs at the inputs of the MAC units. For example, as shown in Fig. 2.3, the configuration of 8b-MAC to 4b-MAC leaves a portion of the MACs unused. The unused region can be masked to reduce the switching activity. The reduced precision MAC (4b or 2b) will have a shorter critical path than the full precision MAC (8b). The shorter critical path can help to increase the operating frequency or to reduce the input voltage for energy efficiency. With sub-word parallel processing, one MAC unit at full precision (8b) can be configured to produce more than one MAC unit of lower precision. As seen in Fig. 2.3, one 8b-MAC can be configured to two 4b-MACs or four 2b-MACs. At constant throughput, the sub-word parallel processing helps to reduce the operating frequency (1 MAC/clock at 8-b precision, 2 MACs/clock at 4-b precision, and 4

MACs/clock at 2-b precision). The reduced switching activity, frequency, and voltages have been explored to increase the overall energy efficiency in the DVAFS. The energy efficiencyy is further improved by modulating the body bias (BB) in an FDSOI technology [16]. The body bias permits tuning of the dynamic vs. leakage power balance while considering the computational precision. On average, 0.26-10 TOPS/W peak efficiency is reported (implemented in 28nm FDSOI technology). Note that processing at the full precision (i.e., 8-bit) with DVAFS comes at a slightly higher energy and area penalty (compared to 8-bit standard precision) due to additional control circuitry for configuration and more extensive register.

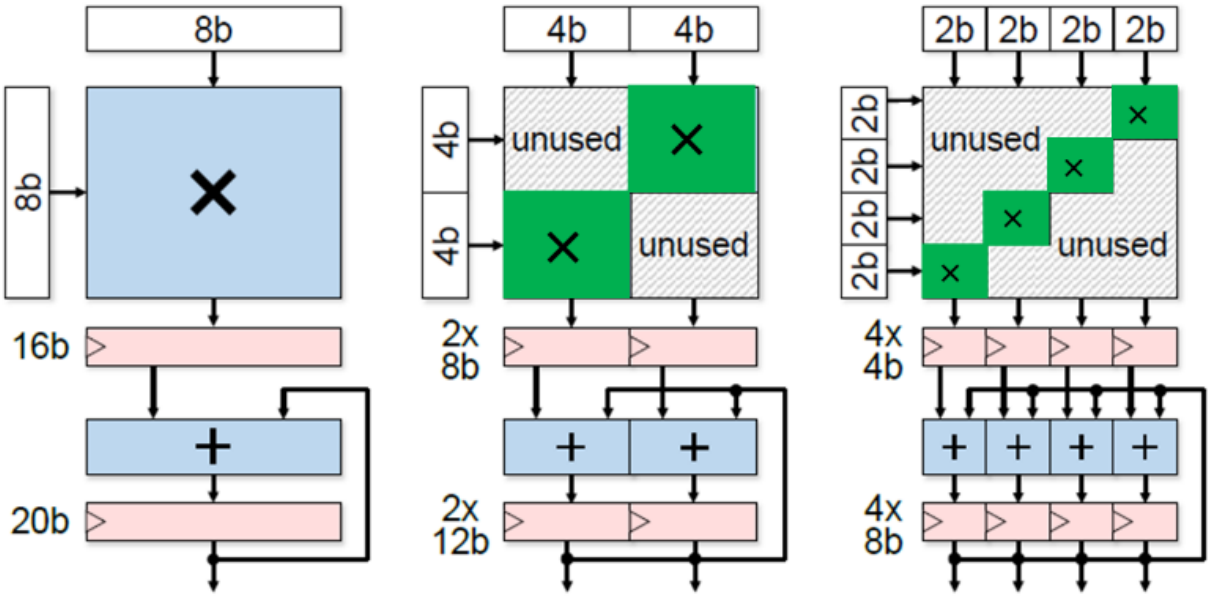


Figure 2.3. Implementation of symmetric precision-variable MAC unit using the DVAFS architecture. An 8x8-bit MAC can be used to implement two 4x4-bit MACs or four 2×2-bit MAC units [16].

Shin et al. [50] proposed a Deep Neural Processing Unit (DNPU) architecture for general DNN models using reconfigurable MAC with sub-word parallel processing (SWP) approach on one operand. In SWP, parts of bits are processed separately using the lower-precision MACs and results are combined to get full results, as shown in Fig. 2.4. In Figure 2.4(a), both activation (*A*) and weight (*W*) have 8-bit precisions, and *W* is represented as two 4-bit sub-words. The SWP architecture generates 16-bit multiplication output by combining the two sub results. In Fig. 2.4(b), *A* has 8-bit precision, but *W* represents two independent 4b words, and the SWP generates two 12-bit multiplication outputs. In other words, the DNPU architecture allows fixed precision on one operand (*A*) and variable precision on the other (*W*). The DNPU reported 8.1 TOPS/W

energy efficiency (with 4-bit precision) on 65 nm CMOS technology. Although the DNPU architecture exposed the SWP for only one operand, the SWP can be exposed in both operands using a DVAFS-like architecture shown in Fig. 2.3.

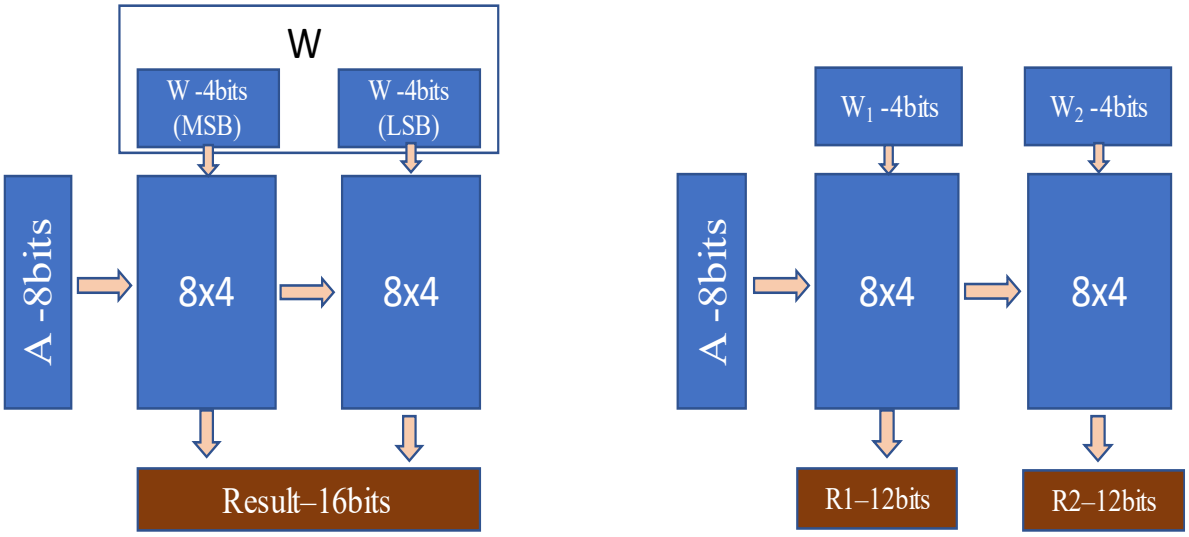
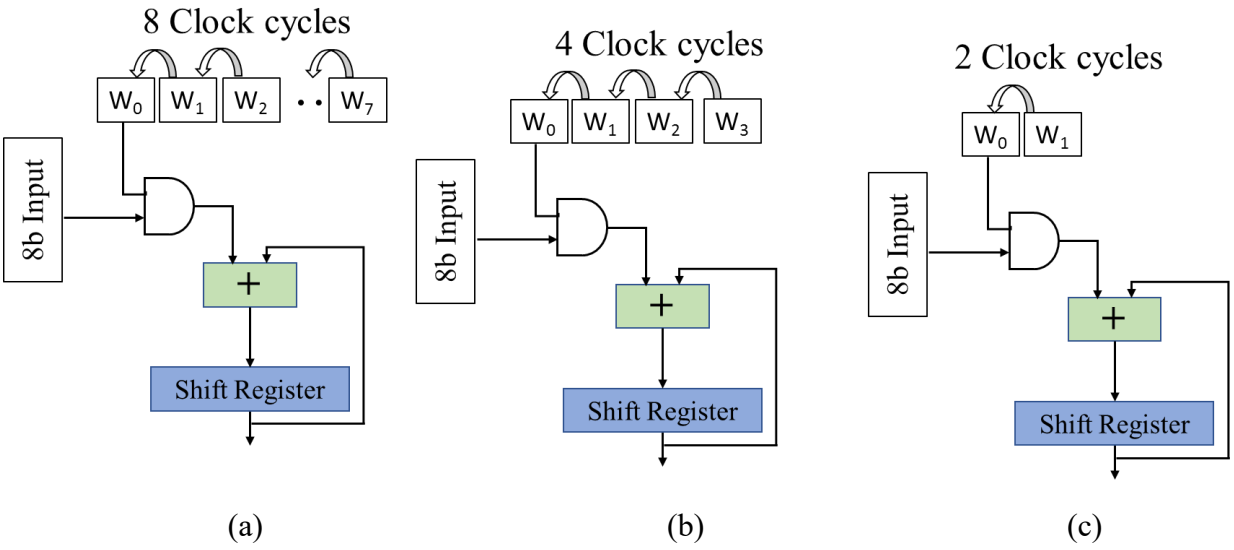


Figure 2.4. Sub-word parallel (SWP) architecture, (a) use of two 8bx4b MAC units to perform one 8bx8b operation, (b) Two 8bx4b MAC operations implemented in parallel.

Lee et al. [51] proposed the Unified Neural Processing Unit (UNPU) architecture using a bit-serial MAC unit. The schematic of a weight-only bit-serial MAC unit is shown in Fig. 2.5. The bit-serial MAC requires just an adder and a shift register and does not require multiplication. In each clock cycle, one bit of weight (LSB bit first) is supplied, and activation is added to the shifted value of the previous cycle partial product. The number of cycles required to finish a MAC operation depends on the weight precision. For an 8-bit precision weight value, eight clock cycles are required to perform the MAC operation, as shown in Fig. 2.5(a). Four and two clock cycles for 4-bit and 2-bit weights respectively, as shown in Fig. 2.5(b) and (c). The architecture supports any weight bit precision from 1b to 16b and reported 1.43× higher power efficiency for a convolutional layer at 4b weight compared to the DNPU.

Alternatively, approximate multipliers or logarithmic multipliers have been proposed to reduce the power and area consumption of multipliers. Note that the neural networks and their associated applications are known for exhibiting intrinsic resilience to errors, which makes them appropriate candidates for approximate computations. A review of the effect of approximate multipliers on the DNN performance can be found in [52]. Ansari et al. [53] proposed an

improved logarithmic multiplier (ILM) that rounds both inputs to their nearest powers of two by using a nearest-one detector (NOD) circuit. The MNIST and CIFAR-10 dataset classification using ILM showed up to a 21.85% reduction in energy consumption and a 1.4% improvement in classification accuracy.



(a) (b) (c)

Figure 2.5. Bit-serial MAC configured as (a) 8bx8b MAC unit, (b) 8bx4b MAC unit, and (c) 8bx2b MAC unit. (Weight-only scaling).

Note that the MAC optimization presented above is primarily based on the binary number system. A few accelerators have been proposed based on non-conventional number systems, e.g., the residual number system (RNS) and posit numbers. Posit numbers have better dynamic ranges and are suitable to represent weights in DNN with lower bit precision. Carmichael et al. [54] proposed a Deep positron architecture based on the posit number system and evaluated its robustness at low precision (< 8 bits). The residual number system is represented by k integers $\{m_1, m_2, \ldots m_k\}$, called moduli which should be relatively prime by each other. In the RNS, an integer value, X, is represented with residues $\{r_1, r_2, \ldots r_k\}$ where $r_i = |X|_{mi}$. Any arithmetic operation in the RNS is equal to the same operation on residues. For example, for two numbers (in RNS) $x_1 = \{a_1, a_2, a_3\}$ and $x_2 = \{b_1, b_2, b_3\}$, x_1+x_2 can be calculated as $\{a_2+ b_2, a_2+ b_2, a_3+ b_3\}$. In RNS, any arithmetic operation can be broken down to the same operation on residues which are represented with lower precision than the actual binary number. It reduces the bit precision requirement at the cost of more computations. In the digital domain, the RNS can improve the speed and reduce the energy in high-precision computations. Olsen et al. [55]

implemented RNS-based matrix multiplication to accelerate neural network processing on FPGAs and achieved 7-9x speed compared to the 32-bit fixed-point implementations. The reduction in the precision requirement is extremely helpful in analog domain implementation, where higher-precision MACs have some limitations with their non-linear and hysteretic behavior. Samimi et al. [56] proposed a RESnet accelerator in the analog domain with RNS. The RNS-based RESnet consumes 145.5× less energy and obtains 35.4× speedup compared to NVIDIA GPU GTX 1080. Accelerators with emerging technologies are discussed further in section 2.3.4.

2.3.2 Dataflow accelerators

The focus of the data flow accelerators is on data management to reduce the off-chip memory bandwidth. Spatial and Temporal architectures are well-studied for data reusability. Efficiency of dataflow accelerators can be characterized with arithmetic intensity, number of operations performed per byte of off-chip memory read. The dataflow can be optimized by reusing the parameters in different layers wherever possible. For example, in a convolutional layer, both the weights and activations can be reused. The each neuron has unique weights in a fully-connected layer, and hence weights cannot be reused but input data (i.e., feature maps) can be reused. The reusable parameters are stored in local registers so that data movement between a MAC and higher-level memory can be reduced.

For a MAC unit, three memory reads (i.e., weight, activation, and partial sum), and one memory write (i.e., updated partial sum) are required. One of the parameters (e.g., weight) can be stored locally in a register file and can be reused for the following few calculations. The parameters stored differ from architecture to architecture based on the data flow structure implemented. There are four major types of data flow structures to manage the input/output data of a MAC in a DNN: No local reuse (NLR), Weight stationary (WS), Output stationary (OS), and Row stationary (RS). In NLR, all memory operations are performed directly from the main memory (e.g., DRAM). In WS, the weights are stored in the RF (i.e., local memory). In OS, the partial sum outputs are stored in the RF to reduce read and write operations. In RS, a row of filter weights is stored in the RF.

Google has developed the Tensor Processing Unit (TPU) accelerator for the efficient implementation of machine learning techniques. The TPU architecture [11] has a systolic array

of 256×256 MAC units as a matrix multiplication unit. The implemented systolic array structure is a 2D single-instruction, multiple-data (SIMD) architecture with specialized weight-stationary dataflow [33]. The block diagram of TPU is shown in Fig. 2.6. The weights can be fetched directly from DRAM and stored in the weight FIFO (First-In-First-Out) register. Input activations from the external memory or previous layer results are stored in the unified local buffer. A systolic data setup block is used to rearrange the input data such that convolution can be performed on a matrix multiply unit. The first version of TPU, known as TPU1, focused on the inference tasks and has been deployed in Google's datacenter since 2015. TPU2, also known as Cloud TPU, has been used for training and inference in the datacenter. TPU2 also adopted a systolic array and introduced vector-processing units.

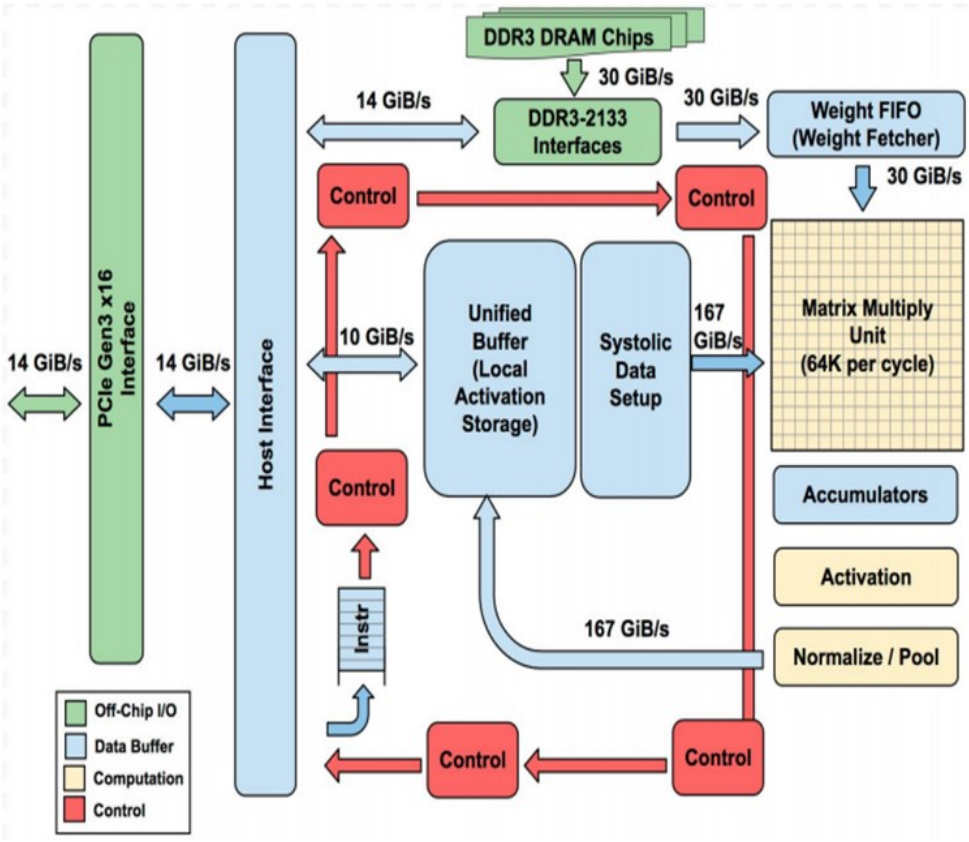


Figure 2.6. Block diagram of a tensor processing unit (TPU) [11].

The SCNN (sparse CNN) accelerator proposed by Parashar et al. [18] uses a dot product dataflow termed as PlanarTiled-InputStationary-CartesianProduct (PT-IS-CP). The Cartesian Product (CP) term indicates the implementation of MAC units in a PE such that a full Cartesian Product of weights and activations ($W \times A$) is calculated. The CP implementation maximizes

spatial reuse. The Input stationary (IS) term indicates that activations are reused at the PEs by storing them in local memory. The Planer Tile defines the distribution of data across PEs. In SCNN, activations and weights are partitioned into smaller tiles and distributed across the PEs.

In the output stationary (OS) dataflow, the partial sums are stored in the local register files. The OS works well with the fully-connected layers, as each neuron output depends on all input activations. Instead of multiplying all inputs with the corresponding weights (which may be a few hundred), in each clock cycle, a few inputs (e.g., K) are multiplied with weights, and the partial sum is stored locally. The entire operation will require N/K clock cycles, where N is the number of inputs. ShiDianNao [48], an example of OS dataflow, was implemented for K=16.

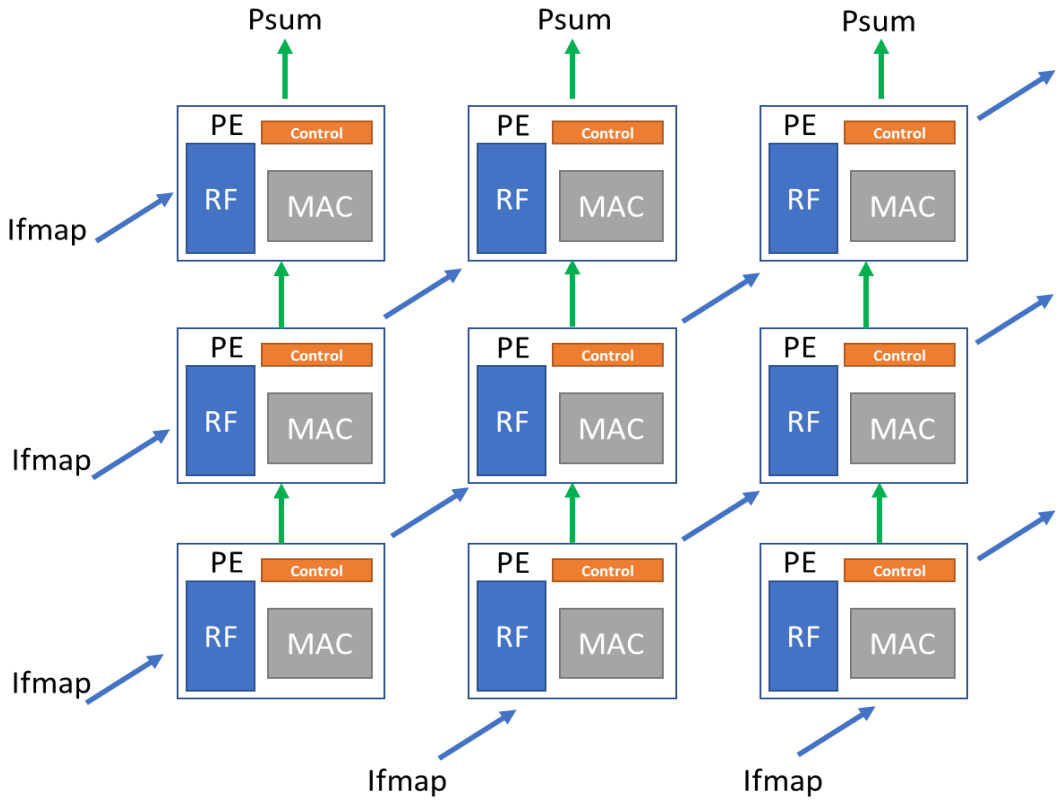


Figure 2.7. Schematic of a row-stationary dataflow.

Chen et al. [57] proposed a row-stationary (RS) dataflow-based accelerator called *Eyeriss* that minimizes the data movement energy on a spatial architecture. Note that in the RS dataflow, a row of operands (i.e., input, weights, and partial sums corresponding to a PE) are stored in the RF. A schematic of a row-stationary dataflow in *Eyeriss* is shown in Fig. 2.7. Inputs are reused across the PEs connected diagonally. The partial sums are accumulated in the vertical direction.

Each PE has local registers to store at least one row of weights and activations, one MAC uni,t and controller. The controller is responsible for the temporal reuse of MAC units to perform 1-D convolution.

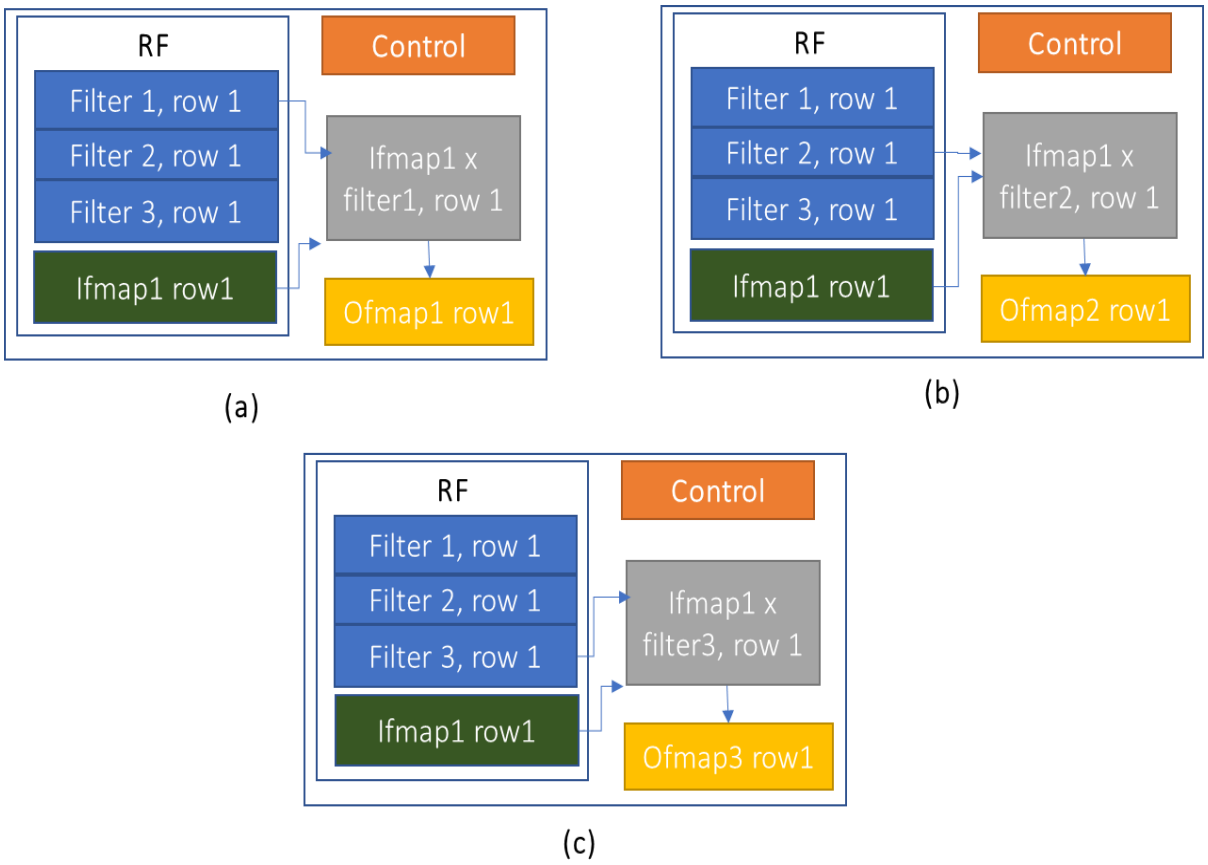


Figure 2.8. Implementation of a row-stationary dataflow on the Eyeriss architecture. (a) 1-D convolution between first row of filter 1(Filter1, row1) and input feature map 1 (Ifmap1). (b) 1-D convolution between first row of filter 2 (Filter2, row1) and input feature map 1 (Ifmap1).

Implementation of 1-D convolution using the RS dataflow in a PE is shown in Fig. 2.8. A sizable portion of RF is allocated to the weights. A row of input vectors is reused to calculate the partial sums of multiple output feature maps. Figure 2.8 shows how the same PE can be used to calculate multiple output features by reusing the input data. It has been shown that the RS dataflow is more energy-efficient than the existing dataflows [57] in both convolutional (1.4-2.5×) and fully-connected layers (at least 1.3× for batch size>16). To support a wide variety of DNN models and further increase in the resource utilization, an improved version of Eyeriss is proposed in [58] called Eyeriss *V2*. The Eyeriss *V2* introduces a highly flexible on-chip network, called hierarchical mesh, which can adapt to different amounts of data reuse and bandwidth

requirements of different data types. Eyeriss *V2* reports 12.6× faster and 2.5× more energy-efficiency than Eyeriss running the MobileNet. Venkatesan et al. [59] proposed multi-level weight-output stationary dataflows: Weight Stationary–Local Output Stationary (WS-LOS) and Output Stationary–Local Weight Stationary (OS-LWS). The advantages of these dataflows over the IS, WS, and OS dataflows are also discussed. An automated framework, MAGNet, to generate an accelerator for a neural network has been proposed in [59]. Using this framework, an accelerator can achieve up to 40 fJ/op and 2.8TOPS/mm^2 in a 16nm FinFET technology.

In most of the DNN accelerators, the layers are processed iteratively. However, by processing each layer to completion, the accelerator must use off-chip memory to store intermediate data between layers as the intermediate data is too large to fit on on-chip memory. Alwani et al. [60] explored the dataflow across layers and proposed the Fused-layer CNN accelerator. In a Fused-layer accelerator, neurons in multiple layers which depend on generated intermediate data are processed once. This increases the data reuse across the layers. The data dependency between the two layers can be seen in Fig. 2.9. Layer 1 output features (Tile 1' and 2') can be further processed to generate layer 2 outputs, which avoids the storage requirement and memory read-write operations for layer 1 output features (Tile 1' and 2'). For example, Tile 1 input data processed through layer 1 generates Tile 1' data. Instead of storing the Tile 1' data in global or external memory, layer 2 computations can be performed to generate the green pixels (layer 2 output). To generate the red pixels at layer 2, only a small amount of data needs to be read from the higher-level memory. The overlapped data can be reused by storing it in the local memories. Fused-layer method avoids the storage requirement of intermediate results (layer 1 outputs) externally. If multiple processors run in parallel, the intermediate results can be reused across the processors without read/write to external memory. Based on this principle, Shao et al. [19] proposed the SIMBA accelerator based on a multi-chip-module (MCM). In the MCM, small chiplets (i.e., small chips) are integrated at the package level. Each chiplet has a 4x4 PE array with weight stationary dataflow. The SIMBA integrates 36-chiplets, each with 4 TOPS peak performance, to achieve up to 128 TOPS peak and 6.1 TOPS/W [19].

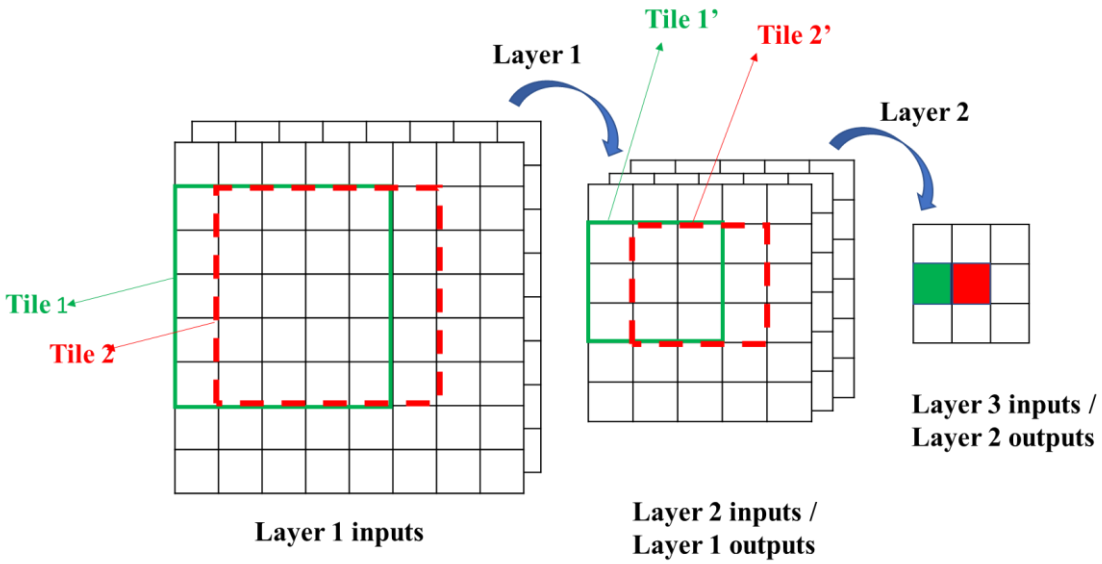


Figure 2.9. Example of fusing two convolutional layers.

2.3.3 Sparsity-based Accelerators.

The computational and memory requirement of a DNN model can be reduced through pruning without significant loss of accuracy. In pruning, at the time of training, any insignificant weights are made to zero. The pruned weights (or zeros) can be in a regular structure or random. In regular structure pruning, also called structured sparsity, a neuron will be removed (i.e., all the weights connected to the neuron are set to zero). The pruning in structure sparsity can be at the level of neuron, filter, or channel of the filter. In unstructured pruning, all the insignificant weights which are random across the weight tensors, are made to zero. The unstructured pruning is simple; it can be done by adding a regularization to the training algorithm. But, due to unpredictable zero patterns in unstructured sparse models, it requires complicated hardware design to compress the non-zero weights and skip zero multiplication. Over time, researcher found complex algorithms for structured pruning where a complete neuron, filter or channel of filters are removed. The architecture for structured sparsity is simple.

Albericio et al. [12] proposed the Cnvlutin architecture to exploit the sparsity in feature maps. Computation with zeros in the inputs are eliminated by indexing the input data. Non-zero input data along with index value are supplied to compute unit. Based on index value, the compute unit selects the corresponding weight from filters and performs multiplication. The

controller fills the index buffer on the fly such that it does not consume extra clock cycle. To further increase the acceleration, Cnvlutin prunes near-zero outputs during inference to increase the sparsity of the next layer's input data. Experiments with several CNNs, including AlexNet, GoogleNet, and VGG-19, showed 1.2–1.6× throughput increases over DaDianNao [47] without any loss in accuracy on ImageNet data. The Cnvlutin reported an area overhead of 4.5% over DaDianNao. Judd et al. [13] proposed Cnvlutin-2 architecture, an extension of Cnvlutin by exploring both input and weight sparsity. Cnvlutin-2 is further optimized to reduce the memory bandwidth.

Eyeriss [15] also explored the sparsity in inputs to reduce energy consumption. MAC units corresponding to the zero inputs are inactivated by a gating method (disable). The gating method saves energy but does not increase the throughput. Eyeriss *V2* [58] can process the sparse data directly in the compressed format for both the weights and activations, and therefore is able to improve both processing speed and energy efficiency with sparse models.

Han et al. [45] deep-compressed the model by pruning the redundant connections and by enabling multiple connections to share the same weight. Deep compression uses threshold-based pruning, quantization, and Huffman coding techniques to reduce the overall size of the model to fit on the chip memory. Han et al. [17] proposed an energy-efficient inference engine (EIE) to accelerate deep compressed model's inference. To exploit the sparsity and reduce the memory bandwidth, the data is compressed using a variation of the compressed sparse column (CSC) format. For each column (M_j) of matrix M, a vector $\boldsymbol{v}$ that contains the non-zero weights, and another equal length vector $\boldsymbol{z}$ that encodes the number of zeros before the corresponding entry in $\boldsymbol{v}$, are stored. Each entry of $\boldsymbol{v}$ and $\boldsymbol{z}$ is represented by a four-bit value. If more than 15 zeros appear before a non-zero entry, then a zero is added in vector v. For example, the following column [0, 0, 1, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 3] is encoded as v = [1, 2, 0, 3], z = [2, 0, 15, 2]. Weight matrix distributed across the PEs and stored in a compressed format. The EIE performs the *sparse matrix* × *sparse vector* operation by scanning vector a (activations) to find its next non-zero value a_j and then broadcasting a_j along with its index j to all PEs. Each PE then multiplies a_j by the non-zero elements in column W_j. Compared with DaDianNao, the EIE has 2.9x, 19x, and 3x better throughput, energy efficiency, and less area respectively [17].

Parashar et al. [18] proposed the SCNN accelerator for compressed-sparse convolutional neural network. Weights and activations are compressed with variants of the CSR methods used in [17]. For example, as shown in Fig. 2.10, multiple 3x3 filters are compressed into a data vector (row-wise), containing non-zero filter values and an index vector. In the index vector, the first value represents the number of non-zero elements in the data vector, followed by the number of zeros before each value in the corresponding data vector. Multiplication between compressed weights and activations is performed like dense matrix multiplication. The output activation's index is calculated based on the inputs and weight's index at accumulation buffers using crossbar connections. The SCNN accelerates a CNN by 2.7x while still being 2.3x more energy-efficient (compared to the uncompressed network).

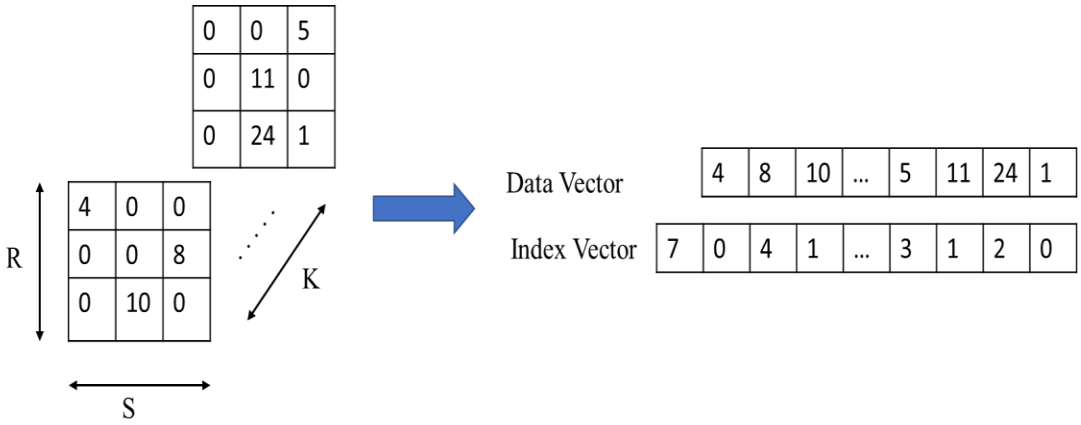


Figure 2.10. Weight compression in SCNN

With zero skipping implementation for sparse models, a small to significant percentage of the MAC units may be end up in the inactive state to synchronize with other PEs. Zhang et al. [61] used parallel associative search to maximize the even distribution of data across the MAC units and implemented it in SNAP accelerator. The SNAP maintains an average of 75% hardware utilization. Similarly, Lee et al. [62] proposed the LNPU architecture for sparse DNN model learning. The LNPU has an input load buffer module which distributes the workload evenly to the PEs, accounting for irregular sparsity. The overall MAC utilization increased in the LNPU. Lin et al. [63] proposed a Dual-core deep learning accelerator based on compression, zero skipping, and Fused-layer techniques.

Zhang et al. [14] proposed the Cambricon-X architecture to exploit the sparsity in filter weights by adding a buffer control module. The buffer control module includes an indexing

module that selects and transfers the useful input neurons (neurons corresponding to non-zero weights in the filter) to PE. A PE stores the compressed filter weights locally and performs the computation asynchronously. Cambricon-X reported a 7.23x speedup and 6.43x energy saving against the DianNao accelerator.

The architectures presented above are unstructured sparsity-based, but the unstructured sparsity in weights needs complex decode module to decompress the weights and calculate the respective activation index. Based on this observation, Zhou et al. [64] showed that the pruning block of weights in a DNN model reduces the irregularity in weight sparsity. Zhou et al. [64] proposed a *Cambricon-S* accelerator that uses structured sparsity in weights and encoded to achieve a higher compression ratio. *Cambricon-S* reported 1.71× speed and 1.37× energy efficiency compared to the *Cambricon-X*.

2.3.4 Hybrid implementation techniques

With increasing complexity in the neural network architectures, the required computing power far exceeds what is achievable with today's technology [65]. Hence alternative technologies like analog computation, photonic and quantum computing are being explored. The new technologies are mainly applied at the ALU level in DNN accelerators to improve the speed and energy efficiency. Therefore, this section (i.e., Section 2.3.4) can be seen as an extension to the ALU-based accelerator classification (Section 2.3.1).

In the ML hardware implementation, the processor-memory bandwidth is often the main bottleneck that limits the achievable energy efficiency. Due to the interconnect loss and signal integrity issues, the data transfer is not as efficient as the data processing. Note that the technologies are optimized for either data processing (processor technology) or storage (memory technology). Therefore, DRAM ICs are used for storage, and processor ICs are used for processing. Bringing them closer through advanced packaging can reduce energy penalties due to the data movement. But Processing-near-memory or Processing-in-memory (PIM) can reduce the data movement. The hybrid memory cube (HMC) technology lets vertical integration of DRAM memories on logic circuits and enables near-data processing. Neurocube [66] and Tetris [67] are two DNN accelerators based on HMC. Given that the DRAM ICs are optimized for data storage, they are few generations behind the logic CMOS devices in terms of computational efficiency. Therefore, analog computation can be an attractive alternative to conventional digital

computation. For instance, multiplication can be directly integrated into the bit-cells of an SRAM array [24].

In recent years, memristors (or programmable resistive elements) show promising performance improvements. In memristors, weight values are stored as the resistor's conductance, and multiplication is performed based on Ohm's law

$$i = G \times V \tag{2-4}$$

where V is the input voltage, G is the resistor's conductance, and i is the output current equivalent to the multiplication result. Fig. 2-11 shows a schematic of the memristor crossbar in which currents in a column are added together. Using Kirchhoff's current law, the resulting current (I) can be expressed as follows.

$$I_j = \sum_k i_{k,j} \tag{2-5}$$

Substituting the i value from Eq. 2-4, we obtain

$$I_j = \sum_k G_{j,k,} \times V_k \tag{2-6}$$

The output current I_j in Eq. 2-6 is equivalent to a neuron output in the neural network. Therefore, the memristor crossbar can be used to implement neural networks. Figure 2.11 shows memristor implementation of a vector matrix multiplication, $(V \times W)$. The digital input X is converted into an analog input V using a DAC converter. The weight values, W, are programmed as the resistor's conductance. The output currents are converted back to the digital domain using ADC converters. The resistive crossbar implementation can reduce the data movement energy.

The resistive memory can be implemented using different technologies such as Resistive RAM (ReRAM), Phase-Change Memory (PCM), floating-gate charged-trap memory, SpinTransfer Torque Magnetic Random-Access Memory (STTMRAM), and Ferroelectric Field-Effect Transistor (FeFET) [68]. ReRAM is a popular technology used in resistive crossbar array implementation for neural networks. The two major limitations of this technology are small tunable conductance range and the parasitic voltage drop across the array. But more importantly, their non-linear and hysteretic behavior limits their usage for applications where higher precision would be needed. Ultimately, to interface with the digital part of the CNN, we need a DAC to

convert the digital inputs to analog voltages and an ADC to convert the summation output voltage/current to digital. The accuracy of such MAC units is limited by the ADC and DAC resolutions as well as other circuit noises. The conductance range and noise levels define the weight precision, and eight-bit weight precision remains at the upper limit using a single non-volatile memory device [68]. The low-precision parameters can still produce similar accuracy in inference, but are generally not sufficient for training. The conductance is always positive, and hence only positive weights can be implemented. For negative-valued weights, w, two weights, w_1 and w_2, whose difference is equal to w ($w = w_1 - w_2$) are implemented, and resulting output currents are subtracted. A few proposed analog NN architectures are PRIME [69], ISAAC [70], Memristive Boltzmann machine [71], Newton [72], PUMA [73], and mCNN [74].

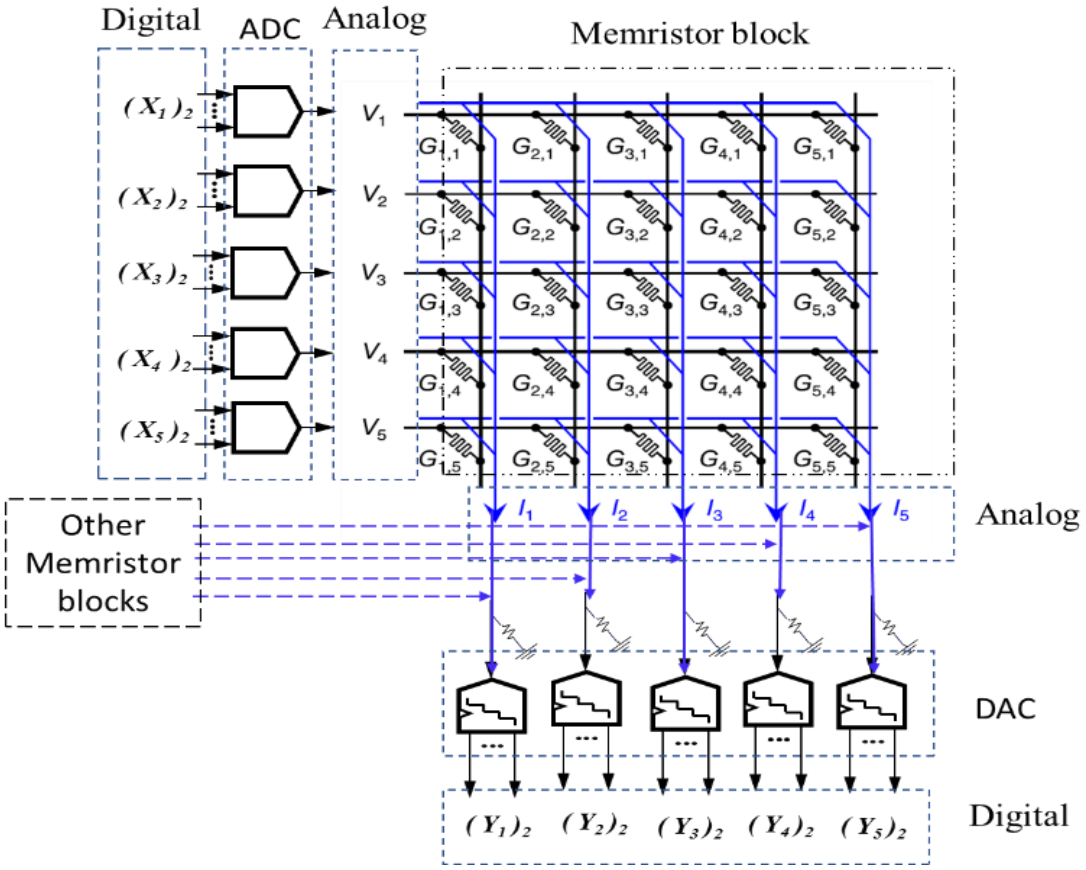


Figure 2.11. Resistive memory crossbar implementing vector-matrix multiplication $Y = X \times G$. V denotes the input voltage vector (analog equivalent of X); G denotes the conductance of memory equivalent to weights, and I denote the resultant output currents (analog equivalent of Y). DAC: Digital to Analog conversion block, ADC: Analog to Digital conversion block.

In some mixed-signal accelerators, the computational units are partially implemented in the analog domain. Cao et al. [75] proposed a hybrid digital mixed signal computing platform

using a Time-Division Mixed signal (TD-MS) multiplier. It usea s 5b TD-MS multiplier and extends to higher precision (6 to 8- bits) using shift and add. Bankman et al. [76] proposed a mixed signal binary CNN processor which performs multiplication in the digital domain and summation using switched capacitor neurons. The weights and input data are represented in binary form hence multiplication in the digital domain is efficient. Detailed reviews on analog neural network accelerators can be found in [68, 77].

Similar to the memristors, the digital data in these analog accelerators has to be converted into analog using DAC before processing in the analog domain. After processing, the result has to be converted back to digital domain using an ADC. The DAC and ADC converters consume more energy with an increase in precision. For higher-precision data, energy consumed by converters can nullify the advantage gained with analog computations and the overall performance may degrade compared to the digital domain. The MAC unit does not always need the ADC or DAC elements, but in most cases, the non-idealities of the analog MAC require digital calibration and correction that mandates ADC and DACs. Like analog accelerators, photonic accelerators are also being explored to enable faster computation with improved energy efficiency [65]. A detailed discussion of such solutions is beyond the scope of this work. However, these solutions also face the same resolution challenges as other analog solutions that limit their usage to certain applications.

2.4 Evaluation

In this section, we present the performance evaluation of a few selected architectures. Most existing research works use measures such as chip area, throughput, latency, and power efficiency for performance comparison. An accelerator proposed for a specific DNN model (e.g., sparsity, kernel size) may not translate its benefits to other DNN models. For example, the performance of sparsity-based accelerators significantly degrades on the denser models due to the presence of additional encode and decode modules. Similarly, the weight stationary data flow typically performs better on the convolutional layers compared to the fully-connected layers because of weight reusability in the convolutional layers. A similarr performance trend is observed in the variable-precision accelerators (ALU-based), where both the latency and the power consumption increase compared to the fixed-precision accelerators on a DNN model running at full-precision. Therefore, it is essential to understand the advantage and drawbacks of

each method of accelerator implementation (e.g., ALU-based, RS, WS, OS, and sparsity-based accelerators).

Parashar et al. [78] proposed a software framework known as *Timeloop* to estimate the energy-efficiency of an accelerator architecture on different workloads without physical implementation. It is claimed that the *Timeloop* framework can give over 95% accuracy compared to the actual physical implementation of hardware. Therefore, the *Timeloop* framework was used to measure the performance of a few architectures. Before considering *Timeloop*, the framework performance on Eyeriss architecture with AlexNet layer 1 workload is verified with manual calculations, the difference is within 5%. Manual calculation uses a similar method proposed by Yang [79]. The workload (AlexNet layer 1) is mapped manually on Eyeriss architecture. The parameters (inputs, weights, and partial sums) are stored across the memory hierarchy (DRAM, global buffer, and RF files) such that minimum data read-write operations (or maximum data reuse) are performed. The final output is written back to DRAM. The number of memories read or write operations of each parameter at all levels of the memory hierarchy are counted. In the calculation, we consider that the 16-bit MAC consumes 2.20pJ per operation (obtained from the *Timeloop* software). Note that the energy consumed for read/write operation at different levels of memory is calculated based on 45nm CMOS technology [78] (DDR4 technology for external memory access). The energy required to read data from RF is assumed to be equal to one MAC operation. The manual calculations require 840 μJ to process AlexNet layer 1 on Eyeriss, and the *Timeloop* reports 866 μJ. The advantage of using the *Timeloop* framework is the optimal workload mapping on an architecture. Therefore, we will be using the *Timeloop* framework to evaluate the performance.

In a DNN model, the size of parameters varies from layer to layer. Let I, W, and O denote the size (in Kbytes) of a convolution layer's inputs, weights, and outputs. In general, $O > I$, W in the initial few layers. This is because a large number of feature maps (typically known as the depth of the layer) are generated at the initial layers. In the later layers, the size of output features O is reduced. Hence, in the last layers, $W >> O$ in general.

The size of parameters can affect the performance of an architecture. Therefore, five different convolutional workloads, which can generalize to a broad range of workloads (with different filter sizes, convolution strides, etc.,) are considered for evaluation in this section. The

configuration of these five workloads is shown in Table 4. The workload, calculated as number of computations in a layer, increases from CONV1 to CONV5.

Table 2.4. Example of five workload configurations in terms of Input (I), Output(O), and Weight(W) sizes. TOTAL-PARAM: Total number of Parameters, I+W+O (in millions). TOTAL-COMPUT: Total number of computations (in millions).

PARA-METERS	CONV1	CONV2	CONV3	CONV4	CONV5
INPUT (*I*)	225X225X3	227X227X3	64X64 X128	17X17 X256	33X33 X96
WEIGHTS (*W*)	5X5X3X96	11X11 X3X96	1X1X128X256	3X3X256X384	3X3X96X256
OUTPUT (*O*)	111X111X96	55X55 X96	64X64 X256	15X15X384	31X31 X256
STRIDES	2	4	1	1	1
TOTAL-PARAM	1.34	0.47	1.6	1.04	0.57
TOTAL-COMPUT	88	105	134	199	212

Using the *Timeloop* framework, the energy performance of three different architectures on the five workloads was calculated. The three architectures considered are Row Stationary (RS), Weights Stationary (WS), and Output Stationary (OS) architectures with the same number of resources (e.g., MACs and memory) available. The allocated resources are based on the existing hardware accelerator *EYERISS* [15]. The availableon-chipp global buffer is set to 128KB and 256 (16x16) PEs, and the local buffer at each PE is 440 bytes. The local buffer is used to store weight, and partial outputs in the RS, WS, and OS architectures. In the RS architectures, the local buffer is partitioned into three parts and is used to store inputs, weights and outputs. In WS, the local buffer is used to store only weights, whereas OS architectures store only partial outputs. The latency is calculated based on the number of clock cycles required to process. The clock frequency is 200MHz (assuming 45nm CMOS technology).

Two performance parameters, latency and energy, are calculated for all three architectures on 5-different workloads using the *Timeloop* framework, and results are shown in Fig. 2.12. In the *Timeloop* framework, the mapper (e.g., a compiler) searches for the optimum workload map on the architecture. The search algorithm requires the performance criteria to select the best match. We choose latency and energy as the optimization criteria. In Fig. 2.12(a), the amount of computation increases monotonically from CONV1 to CONV5, but the energy consumption of the architectures does not always increase with computations. This is because data transfer contributes a significant part of the total energy. For example, between CONV4 and

CONV5, there is a slight reduction in the total energy consumption despite increased computation due to less data transfer. Therefore, the energy efficiency of an architecture depends on both computations and the number of parameters. From Fig. 2.12(b) and (c), it can be observed that the RS architecture has the lowest latency (combining all layers), and the WS architecture consumes the least energy.

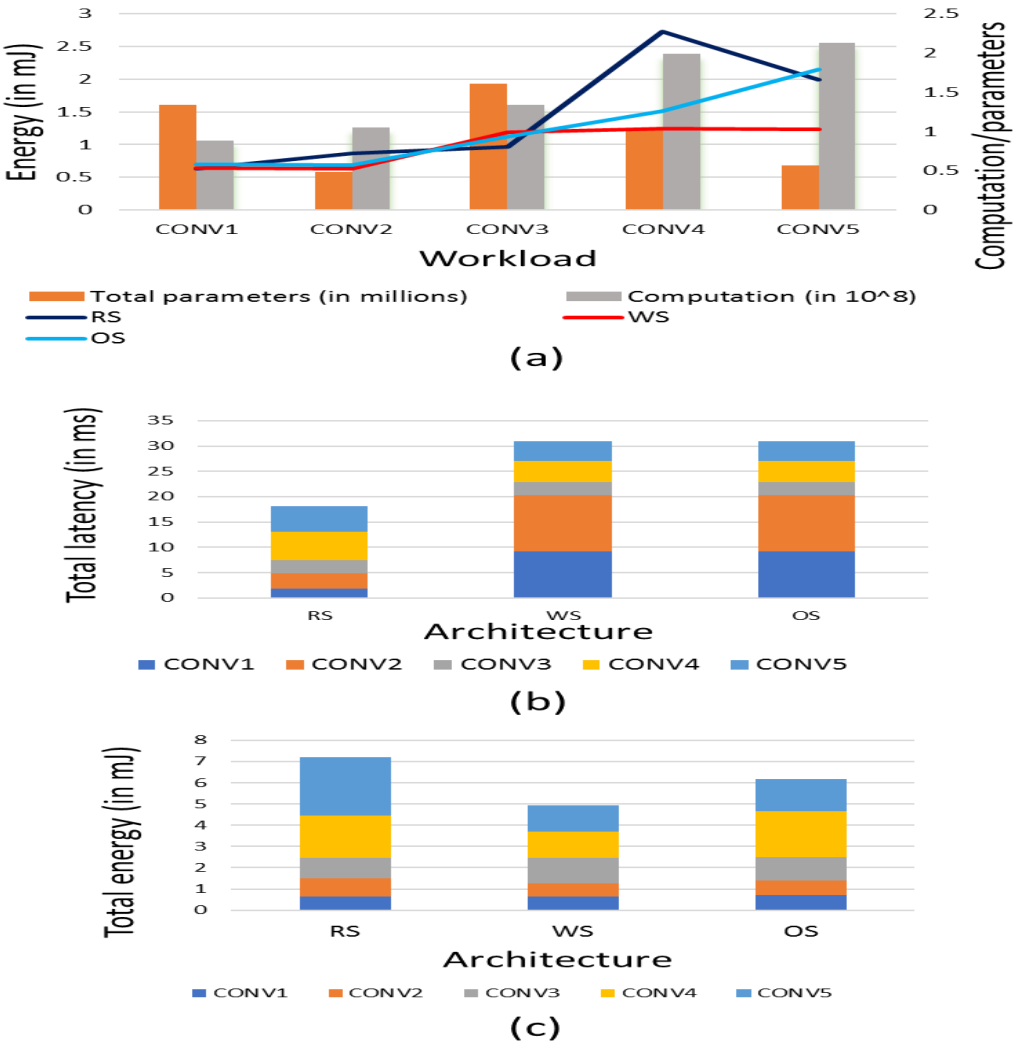


Figure 2.12. Performance of three different architectures: (a) energy consumption in different workloads. (b) architecture latency on all workloads. (c) architecture total energy consumption.

The dataflow efficiency depends on how much the parameters are reused within the local memory once they have been read from the external memory. In the convolution operation, the filter properties (height (R), width (S), and channel (C)) define the data reusability. For example, in a convolution with a 3×3 sized filter, one input can be reused to calculate nine partial products

(with nine weights) corresponding to nine outputs. Therefore, the performance of the RS, WS, and OS dataflows are evaluated with filter size, as shown in Fig. 2.13. In this figure, the workload of CONV5 is varied by changing the filter size from 1×1 to 11×11.

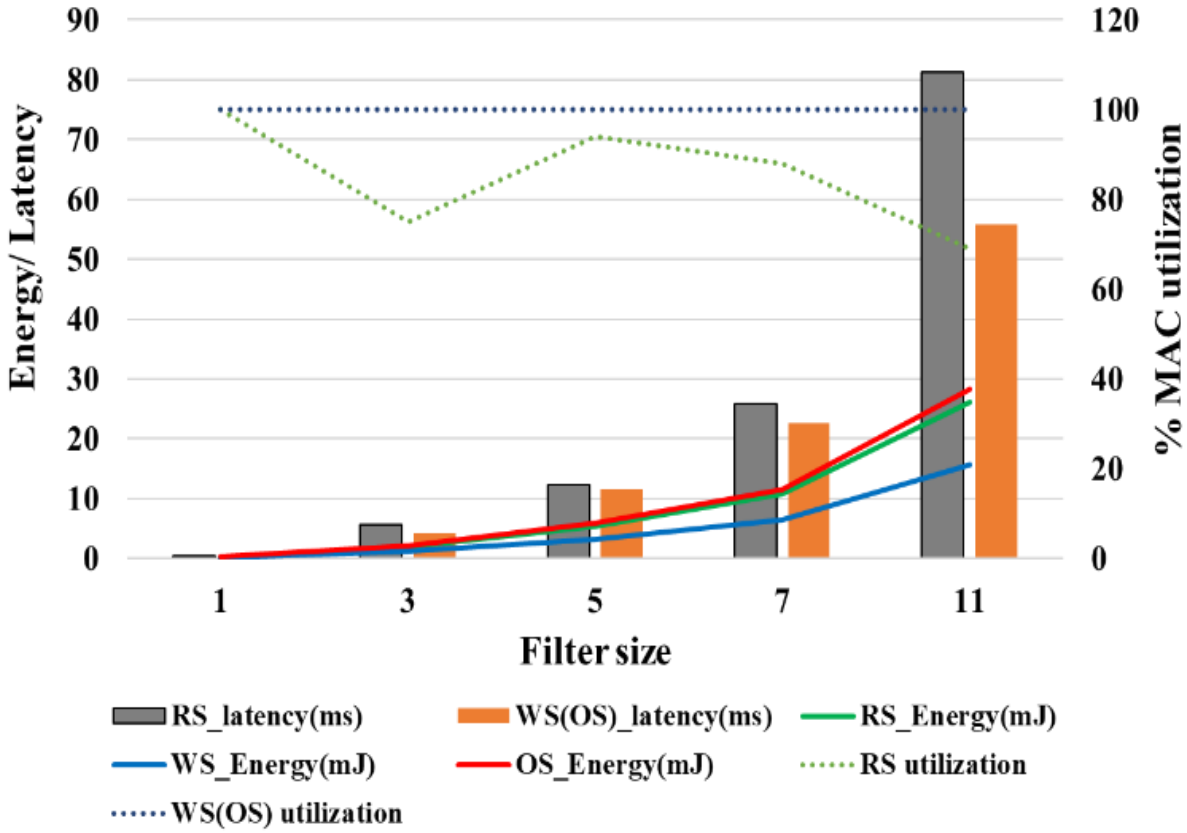


Figure 2.13. The RS, WS, and OS performance with variation in filter size. Note that the Latency and MAC utilization in the OS and WS are same, and their plots coincide (dotted black line).

The total computations increase with the filter size and require more processing energy. From Fig. 13, it can be observed that energy consumption rises with filter size for all the architectures. But the energy consumption of the RS and OS architectures increases more than that of the WS architecture. To understand the energy variation, we looked at the energy consumption of DRAM, global buffer, local buffer access, and MAC unit per computation. The MAC unit and local buffer consume similar energy across the filters. The DRAM and global buffer access energy varies with filter size, as shown in Fig. 2.14. For filter 1x1, the WS and RS architectures consume a similar amount of energy (as seen in Figs. 2.13 and 2.14) because when the filter size is 1x1, the weight reuse is identical in both the architectures. The input data reusability increases with filter size but requires more local memory to store the filter weights. In

the WS architecture, the local memory is allocated primarily for filter weights and can keep all the weights for even large-size (11x11) filters. Therefore, the DRAM access energy per computation decreases for the WS with increasing filter size, as shown in Fig. 2.14. The local memory is primarily allocated for partial products in the OS architecture and shared with all three parameters in the RS. The OS and RS architectures may not have sufficient space in the local memory for large-size filters, and hence the increased data reusability with filter size does not significantly affect the DRAM access energy, as shown in Fig. 2.14. The WS architecture requires less DRAM and global memory access energy, which means it maximizes the data reuse within the local memory and requires less access to the higher-level memory. Therefore, the WS consumes less energy among all architectures with filter size increase, as shown in Fig. 2.13.

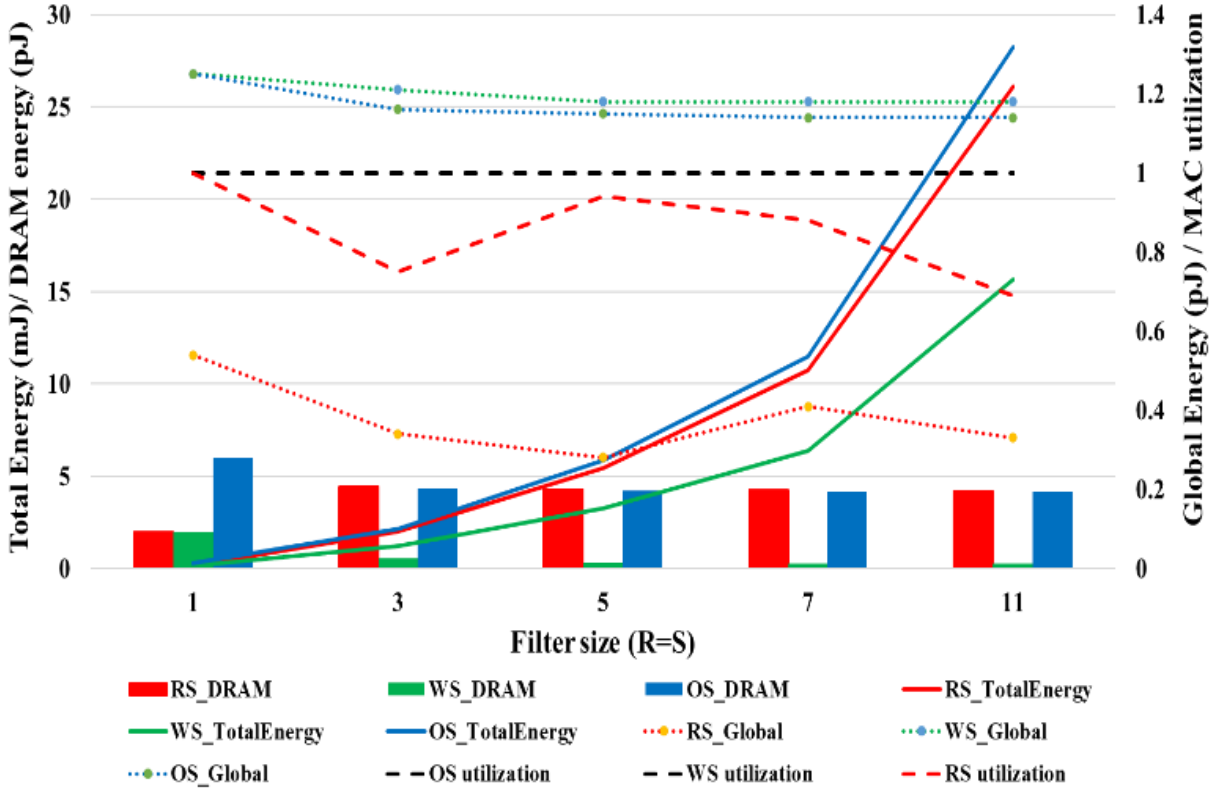


Figure 2.14. Energy consumption and MAC utilization in the WS, OS, and RS architectures for different filter sizes. R and S are the number of rows and columns, respectively.

The MAC utilization of the RS varies with the filter size, as shown in Fig. 2.14 (see the dotted lines). In RS architecture, the PEs are connected in such a way that the inputs are reused in diagonal PEs, and partial sums are accumulated across vertically connected PEs, as shown in Fig. 2.7 (the directions can be configured). When mapping the workload on the PE array, a few

PEs may end up unallocated. For example, in mapping a 3x3 filter on four PEs, three rows of filters can be stored in three PEs and accumulate the partial products to get the convolution output. The fourth PE is unused and can be used to calculate the next output, but the partial product must be stored in the memory and be read in the next cycle. The additional energy required for the partial product memory read-write can defeat the advantage of using the fourth PE. Therefore, only three PEs are used for calculations, and fourth one is left ideally. Due to only three PEs being utilized effectively, more clock cycles are required to complete the convolution. To fully use the PEs in the RS dataflow, the array size should be multiples of the filter size. In this experiment, the PE array size is 16x16, which is not multiples of 3, 5, 7, 11 (i.e., the filter size). Therefore, the MAC utilization of RS dataflow varies with filter size (see the dotted green line in Fig. 2.13). The decrease in MAC utilization increases the latency as, shown in Fig. 2.13. For filter sizes from 5 to 11, the latency of the RS increases more compared to the WS or OS because of the drop in the MAC utilization. The latency difference between the RS and WS/OS is small at filter size 5 (~0.8) compared to the difference at filter size 3 (~1.4) because of the increase in the MAC utilization for RS.

The convolution stride is another parameter that can affect the data flow. With a stride of greater than one, the input features may not be reusable in two neighboring output feature (pixels) computation. For example, with the stride of one, six input pixel values (i_{23}, i_{33}, i_{43}, i_{24}, i_{34}, i_{44}) out of nine can be reused for the next window, as shown in Fig. 2.15(a). In the convolution with a stride of two, only three input pixel values (i_{24}, i_{34}, i_{44}) out of nine can be reused for the next window, as shown in Fig. 2.15(b). On the other hand, in the convolution with a stride of three, there are no common pixel data in the consecutive windows, as shown in Fig. 2.15(c). Therefore, the input data reusability depends on the stride value, and there is no reuse of input data in consecutive output feature calculation if the stride value is more than the filter size.

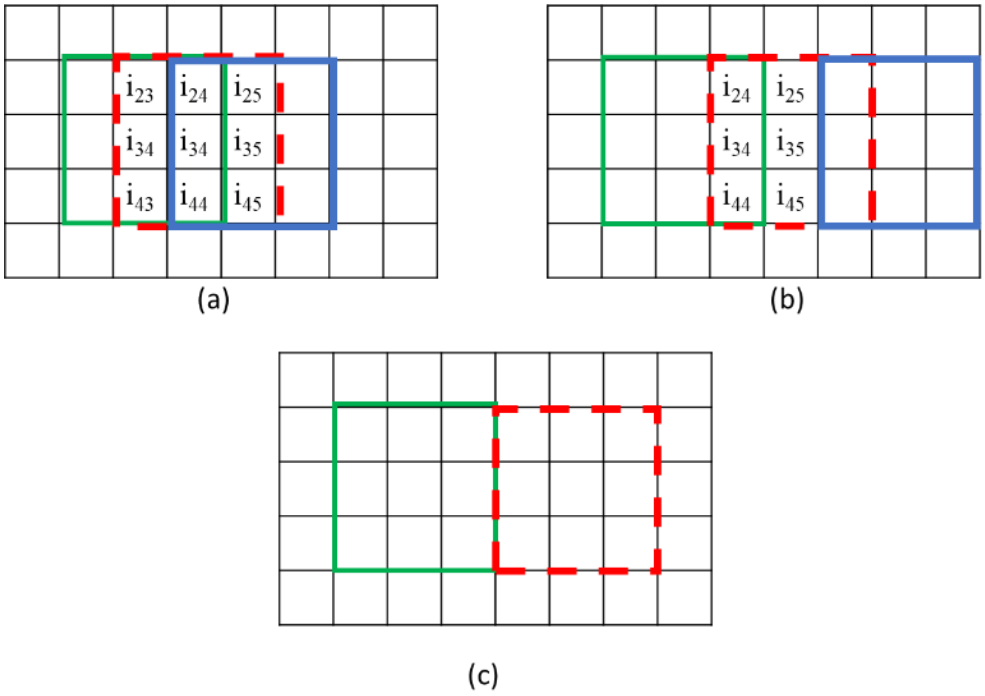


Figure 2.15. Impact of convolution strides on the input data reusability. (a), (b), (c) represent input feature maps with filters (colored boxes) imposed on it to show consecutive convolution windows with stride values of 1, 2, and 3, respectively.

Note that it is important to understand dataflow architectures' efficiency with stride variation. In this experiment, the stride size is varied from 1 to 4 in the CONV4 workload. The total energy consumption depends on the type of workload and filter size, as shown in Fig. 2.12(a) and 2.14. Hence, instead of comparing the total energy, we compare the energy normalized to stride one in respective architectures. The normalized energy here indicates the change in energy due to the change in the stride value. The DRAM access energy increases by about 120% in the WS architecture as stride value changes from 1 to 4, as shown in Fig. 2.16. The DRAM access energy changes because of the change in the input data reusability and may require reading the full window of input data at each cycle, as shown in Fig. 2.15. In the RS architecture, a row of inputs is stored in the connected PEs and used in the later computations, if not in consecutive computations. Therefore, the DRAM access energy changes more in the WS compared to the marginal change in the RS. The total energy changes with DRAM access energy, more than 20% increase in the WS, and marginal change in the RS, as shown in Fig. 2.16. Therefore, for large stride values, the RS has better data reusability compared to the OS and WS architectures.

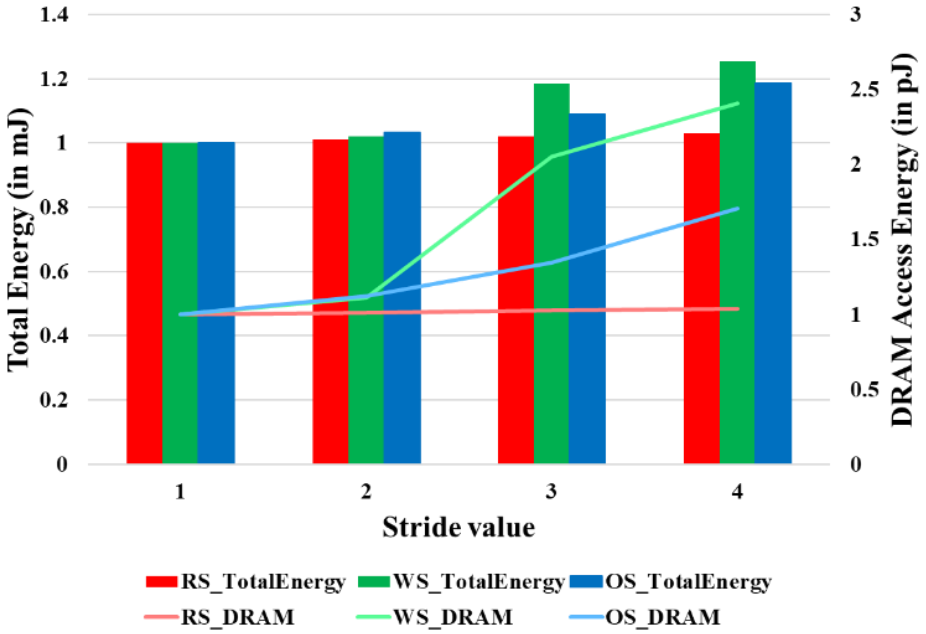


Figure 2.16. Energy variation in the RS, WS, and OS with stride variation for CONV4 workload.

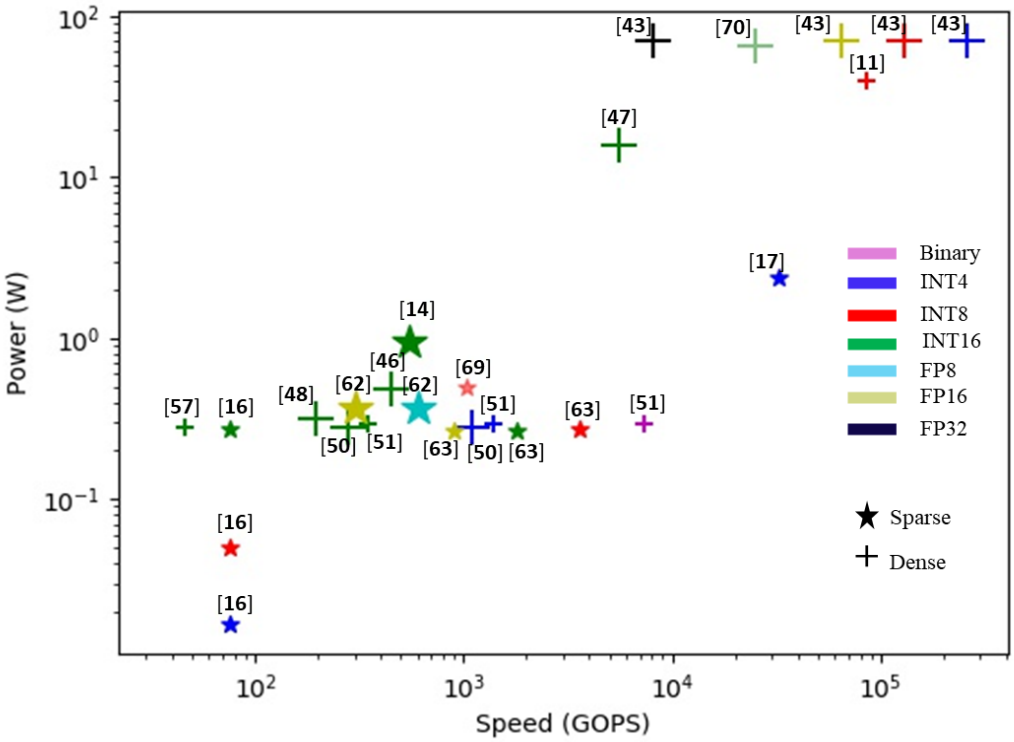


Figure 2.17. Performance of DNN architectures with different precisions and sparsity levels. The sparsity-based accelerators are denoted with star marks and dense models with plus sign. Small size mark indicates real-time performance, and large size mark indicates peak performance.

Note that most accelerators reviewed in the previous section vary in terms of the available MAC units, memory size, dataflow implemented, and workload used. Therefore, energy and latency parameters are insufficient to evaluate or compare the existing accelerators. It is difficult and time-consuming to implement all existing accelerators on the *Timeloop* framework, keeping the same resources, to obtain the performance metrics for comparison purposes. In the ALU-based accelerators, the MAC implementation enables performance improvement. Hence, the ALU-based accelerators can be evaluated by comparing single MAC units. Camus et al. [30] analyzed precision scalable MAC units from different accelerators. Similarly, the performance improvement in sparse-based accelerators is defined by the sparse encoder and decoder modules. To observe the trend in ALU and sparse-based accelerators, the power vs speed plot of a few accelerators is shown in Fig. 2.17. The data for Fig. 2.17 has been obtained from a standard repository [80]. In the Fig. 2.17, the small size marks indicate the real-time performance, and the large size marks indicate the peak performance. The ALU-based accelerators evaluated for at least two precisions are considered. From Fig. 2.17, it is observed that the sparsity-based accelerators consume less power (star marks in Fig. 2.17). The advantage of sparse architectures depends on the sparsity in the input data. For highly sparse data, the additional cost of encoder/decoder can be overcome by the computational advantage (i.e., smaller number of MAC operations after zero skipping). Running a dense model on a sparse accelerator can degrade the performance. Therefore, it is crucial to evaluate the sparse accelerator with varying sparsity (e.g., from 5% to 90%). Precision can also affect the power and speed of an architecture. Low-precision accelerators provide high speed at lower power (see the blue and red color marks in Fig. 2.17). In sub-word parallel architectures, by running at half-precision, the speed can be doubled at the same amount of power ([32], [33], [58]) or power can be reduced at the same speed ([31]). The binary and INT4 precision architectures can achieve high speed at low power but have limited applications.

2.5 Conclusion

Understanding the factors affecting the performance of an DNN accelerator is vital to develop an energy-efficient accelerator. In this study, three major areas ALU, dataflow, and sparsity are identified as potential areas to improve the overall performance of a DNN accelerator. The existing architectures were classified into four categories. The advantages and

drawbacks of each category are discussed. A variable precision ALU can take advantage of sub-word parallel processing for low-precision DNN models to improve overall throughput or to reduce the power requirement. But a precision-variable ALU comes with a complex configuration circuit. An efficient data flow can improve the arithmetic intensity and memory bandwidth requirement. The dataflow efficiency can vary from layer to layer or with filter size. The sparse models can reduce the power requirement by skipping zero multiplication but increase the latency per MAC operation. Three dataflow architectures are evaluated. The dataflow efficiency depends on the workload. Hence, the dataflow must be chosen based on the accelerator application. The classification is discussed in Section 2.3. will help the readers in selecting the best technique at different levels in architecture. An efficient DNN accelerator should have a precision-variable ALU, flexible dataflow for all types of layers in a DNN model, and explore the sparsity with simple control circuitry.

Chapter 3

An Iterative MAC Model

In the previous Chapter, we identified three key areas: arithmetic logic unit, data flow, and sparsity that have the potential to improve the overall performance of a hardware accelerator. In addition, hybrid (mixed analog and in-memory) architectures were discussed for high-speed implementation. The efficiency of the dataflows was evaluated. In this Chapter, we further explore the arithmetic logic unit architectures. It is generally known that the training of DNNs requires higher precision compared to inference. As a result, few hardware accelerators support DNN training. In this Chapter, we propose a low-precision iterative MAC unit-based accelerator intended for inference which can also be used to train the DNN model. The proposed MAC unit can provide good performance and flexibility.

3.1 Introduction

As observed in Chapter 2, the DNN models have been modified by researchers to optimize the performance of hardware accelerators, such as throughput, latency, memory, and power requirements. The two most popular techniques are parameter quantization and parameter reduction. In parameter reduction, the number of distinct weights in a DNN model is reduced. Parameter reduction utilizes different compression techniques and reduces on-chip memory requirements [45]. Parameter quantization leads to a reduction in bit length (precision) to represent weights and activations. The DNN models are typically quantized to fixed-point to implement on embedded systems. The fixed-point arithmetic unit takes the least number of resources. Therefore, various DNN accelerators have been proposed based on fixed-point ALUs. However, problems arise when offline training is required. Existing fixed-point-based accelerators do not support training. In this Chapter, we look at fixed-point ALU implementations for inference that can also train the DNN model.

3.2 Precision requirements

The training phase requires calculation and backpropagation of the error gradients through the network. The weights (W), error gradient (δ), and weight gradients (ΔW)

generally have a large dynamic range during the training. Therefore, the training of a DNN model is typically done in single precision (i.e., 32-bit Floating-point, FL32) format. However, in the inference mode, the model parameters do not change. The range of weights is known before the deployment of the DNN model. For example, Fig. 3.1 shows the weight distribution of two convolutional layers, conv1 and conv3, and two fully-connected layers, fc7 and fc8, of AlexNet [81]. Note that the weights are small in value and have a normal distribution (a similar kind of variation is observed in many standard networks, such as VGG-16, GoogleNet, and ResNet). The weights of a DNN model typically follow a normal distribution. Inside a DNN, the mean and variance of the distribution vary from layer to layer. To represent weights from such distributions, FL32 format is not necessary. Lower bit width data formats can effectively represent such weights. Bit length requirements for DNNs can be greatly reduced with only slight variations in the accuracy. FL32 (32-bit length) weights can be represented in FL8 (8-bit length) without loss of DNN precision [81].

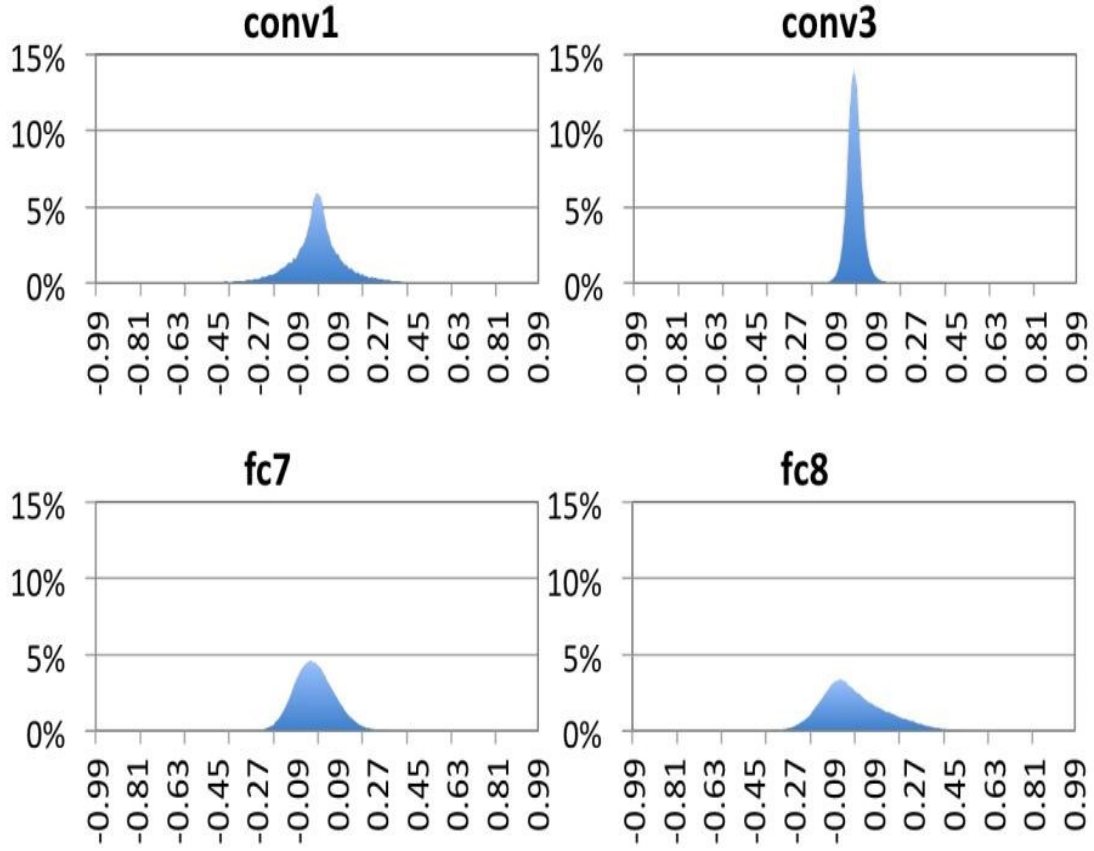


Figure 3.1. Weight distributions in four different layers of AlexNet. conv1 and conv3 are convolutional layers and fc7, fc8 are fully-connected layers [81].

There are different ways to quantize the model parameters. One simple way is to use fixed-point representations with enough bit width. The fixed-point operations do not need any conversion or scaling operations to obtain the result. Hence, a fixed-point arithmetic unit can provide the lowest latency with less power consumption within a smaller area. In dynamic fixed-point format, the binary point position is arbitrary and requires scaling after an arithmetic operation. Other quantization approaches include binary, ternary, lower bit-width floating, and non-linear (e.g., logarithmic) quantization. The binary and ternary representations of parameters lead to better hardware performance but degrade accuracy and model capabilities [44]. Non-linear quantization reduces the bit lengths but requires relatively complex arithmetic hardware units. Hence, fixed-point hardware is preferred inside an accelerator.

The accuracy loss of AlexNet with different quantization formats is shown in Table 3.1. The weights and activations are quantized to different bit lengths, and the accuracy drop with respect to FL32 representation is shown in Table 3.1. It is observed that the reduction of bit length from 32-bit to 8-bit results in a marginal loss of accuracy [22, 45, 83]. Binarization (i.e., 1-bit representation) of AlexNet results in an accuracy loss of more than 20% which is undesirable [87]. Non-linear quantization can produce similar accuracy as FL32 but makes the MAC unit design complex. The deep compression model [45] can reduce the weights to 8 bits, and activations to 16 bits without affecting the accuracy. Dynamic fixed-point quantization can also produce good accuracy but requires scaling after an arithmetic operation [22, 83]. Quantization-aware training approaches [82] have been proposed in the literature, which can reduce the accuracy loss arising due to quantization.

A well-optimized DNN model may require different bit lengths at different layers within the model. For example, the optimized bit length requirements for a five layers Convnet (three convolutional and two fully-connected layers) are 8-7-7-5-5 bits, respectively [42]. In other words, no standard precision is likely to be optimal for all the layers or models. Therefore, a flexible DNN hardware accelerator (or the associated MAC units) should be able to support all possible bit precisions.

Parameter quantization helps in implementing pre-trained DNN models in embedded systems. However, problems arise when on-the-job training is required. The model might need training with domain-specific or confidential data for performance optimization. The weight gradients generally have small values (in a large range) and calculating these values may require

high-precision ALU units. Micikevicius et al. [92] used mixed-precision to train a network. The weights, activations, and gradients are stored in IEEE half-precision (FL16) format, and a master copy (original) of weights are stored in single-precision (FL32) format. The weight gradients are added to the master copy of weights after each iteration, and then the updated weights are converted to half-precision for the next iteration. The approach works for various models, including convolutional neural networks and recurrent neural networks.

Table 3.1. Different methods to reduce numerical precision for AlexNet, accuracy measured for TOP-5 error on IMAGENET data [24].

Reduction Strategy		**Bit width**		**Accuracy loss vs. FL32 (%)**
		Weights	**Activations**	
Dynamic Fixed-point	Without fine-tuning [83]	8	10	0.4
	With fine-tuning [22]	8	8	0.6
Fixed-point Quantization (only weights)	Binary Connect [84]	1	FL32	19.2
	Binary Weight Network [84]	1*	FL32	0.8
	Ternary Weight Network [41]	2*	FL32	3.7
	Trained Ternary Quantization [85]	2*	FL32	0.6
Fixed-point Quantization (both weight and activations)	XNOR-Net [86]	1*	1*	11
	Binarized Neural Networks [87]	1	1	29.8
	DoReFa-Net [88]	1*	2*	7.63
	Quantized Neural Networks [37]	1	2*	6.5
	HWGQ-Net [89]	1*	2*	5.2
Non-linear Quantization	LogNet [90]	5 (conv) 4 (fc)	4	3.2
	Incremental Network Quantization [91]	5	FL32	-0.2
	Deep Compression [45]	8 (conv) 4 (fc)	16	0
		4 (conv) 2 (fc)	16	2.6

* Quantization is not applied to first and/or last layers

Wang et al. [93] trained AlexNet with ImageNet data using FL8-bit format weights and activations except for the first and last layers. In the implemented MAC unit, multiplication is done in FL8, and accumulation in FL16 registers. A speedup factor of 2~4× is achieved without any loss in accuracy. Das et al. [94] proposed a shared exponent representation of tensors and developed a Dynamic Fixed-Point (DFP) scheme, which uses INT16 tensors with a shared 8-bit tensor-wide exponent to represent the parameters. It uses an Integer Fused-Multiply-and-

Accumulate (FMA) unit for computation. An overall 1.8× speedup in training throughput is achieved compared to the baseline FL32 performance.

It has been shown [81, 92, 93, 94] that the training phase requires higher bit-precision compared to inference. Therefore, an accelerator designed for training is generally inefficient if also be used for inference. Accelerators designed for training consume more power and less throughput than those intended for inference. Consider the backpropagation equations used for the network training as given below:

$$\delta^{l} = [W^{l+1}\delta^{l+1}] \times f^{'}(v(n)) \tag{3-1}$$

$$\Delta w_{ji}^{L}(n) = \eta \times \delta_{j}^{L}(n) \times x_{i}^{L-1}(n) \tag{3-2}$$

The computation of the local gradients (δ^{l}) involves at least one forward pass parameter, i.e., the weight (W^{l}) and/or activation (x^{l-1}). The $f^{'}(v(n))$ of most widely used ReLu function inside a DNN model is 1. The weight and activation parameters used in the local gradient calculation are the same values used in the forward pass (inference mode). This suggests that the gradient parameters may need higher precision, but one of the operands (e.g., W^{l+1} in Eq. (3-1)) involved in gradient calculation can be represented in lower precision. Therefore, a high-precision multiplication unit is not necessary for training. For example, assume that the forward path is calculated in 8-bit integer (INT8) representation, meaning the weights and activations are represented in INT8. In backpropagation, even though gradients are represented in FL16 format, a full FL16×FL16 multiplier is not needed. A FL16×INT8 multiplier unit is enough. Further, if we can represent FL16 with an 8-bit mantissa and 8-bit exponent, then the multiplication can be simplified as ((8-bit mantissa) × INT8) *exponent. This way, the floating-point multiplication can replace with a lower-precision fixed-point multiplier. Replacing the floating-point multiplication with fixed-point in the hardware improves the latency and power efficiency exponentially.

Apart from the above proposed precision models, researchers around the world are exploring different solutions to the training step using the integer representation of gradients, without affecting the convergence. In this Chapter, we propose a flexible iterative MAC unit optimized for inference that can be used to train a DNN.

3.3 Iterative MAC Unit

It has been shown in the literature [16, 50, 51] that a high-precision MAC unit can be implemented using lower precision MAC units. In this section, we use this approach to design a low-precision MAC unit that can be used for DNN inference and can be used recursively to train a DNN. A 16×16-bit multiplication can be implemented using four 8×8-bit MACs in parallel or one 8x8-bit MAC in serial, as shown in Fig. 3.2. Figure 3.2(a) shows the parallel implementation, all the four partial products are calculated in parallel and scaled results are added. Figure 3.2(b) shows the serial implementation, one 8-bit multiplier is reused in time to calculate four partial products. The parallel method consumes more resources and space but has less latency (single iteration) whereas the serial method consumes fewer resources but has higher latency (four iterations).

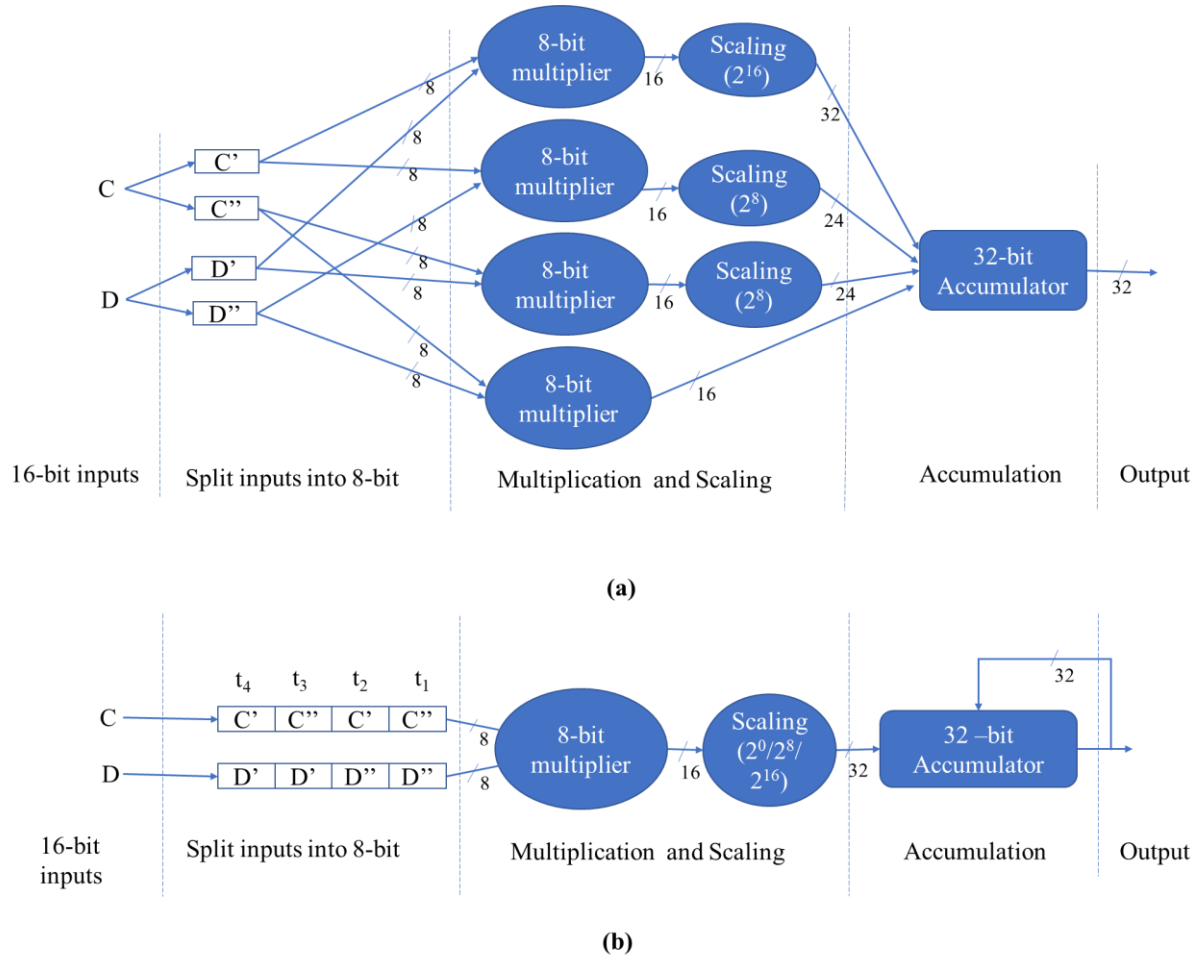


Figure 3.2. A 16-bit MAC implementation. (a) Parallel implementation using four 8-bit MAC units, (b) Serial implementation using a single 8-bit MAC.

It is possible to obtain identical outputs by the two approaches in Fig. 3.2. However, in the serial method, it may be possible to reduce the computations if some error in the output calculation is allowed. For example, consider two INT16 numbers: $C = 6,244$ and $D = 3,272$. The number *C* and *D* can be represented by two INT8 numbers as follows

$$C = C^{'} \times 2^{8} + C^{"}$$
$$D = D^{'} \times 2^{8} + D^{"}$$

here $C^{'}, C^{"}, D^{'}, D^{"}$ are 8-bit numbers. $C^{'}, D^{'}$ are MSB 8-bits and $C^{"}, D^{"}$ are LSB 8-bits. For $C = 6,244$, $C^{'} = 24$, $C^{"} = 100$. Similarly, for $D = 3,272$, $D^{'} = 12$, $D^{"} = 200$. The exact result $C \times D$ is 20,430,368. If the multiplication is performed using just the upper bytes, $C^{'}$ and $D^{'}$, and scaled appropriately, we will obtain $(C' \times D') \times 2^{16} =$ 18,874,368, which is 92% of the exact result. Further, calculating the two more partial products $C^{'} \times D^{"}$ (=1,228,800) and $C^{"} \times D^{'}$ (=307,200) and adding it to the first product with appropriate scaling (2^{8}), the result is 20,410,368, which is 99.9% of the exact result. The calculation of the lower byte partial product (4^{th} partial product) can be skipped if a 0.1% error is acceptable in the final value. In other words, with the allowance of error in the final output, some partial products can be omitted.

As the DNN models are error resilient up to a certain level, we can reduce the number of partial products required to be calculated. The parallel implementation is not advantageous because all four partial products are calculated simultaneously. In the serial implementation, if we can calculate the partial products one after another, starting from the product of upper bytes, we can eliminate partial product calculation using lower bytes. Hence, fewer calculations result in less power consumption and low latency. The serial implementation can also be extended to any higher-precision MAC operation without any additional multipliers. The serial implementation with a threshold limit for lower partial products calculation is called an iterative MAC unit. In the iterative MAC, the upper byes' partial product is calculated first. The iterative MAC has the required flexibility and can take advantage of error resilience to reduce the latency and power consumption of the hardware.

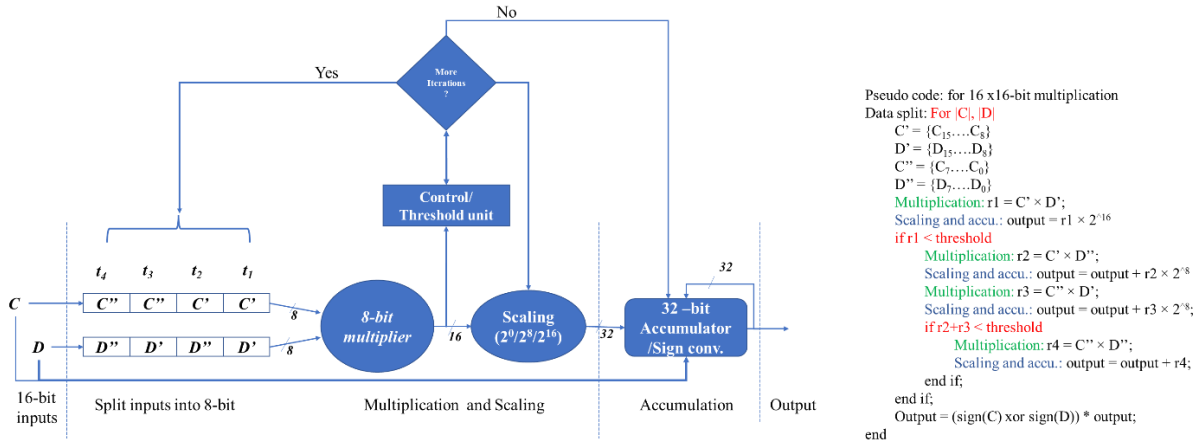


Figure 3.3. Proposed 8-bit iterative MAC implementation for 16×16-bit multiplication.

An 8-bit iterative MAC unit is shown in Fig. 3.3. Partial product using upper bytes is calculated first. The partial product is compared with the set threshold value. If the partial product exceeds the threshold value, no more iterations are calculated. The scaled partial product is the approximate output. If the partial product is less than the threshold value, then the next partial product is calculated and compared with the threshold. The process is continued until the threshold condition is satisfied or all the four partial products are calculated. For a 16-bit multiplication, minimum one iteration and maximum of four iterations are required. To optimize the usage of hardware resources, it is crucial to know when all partial products should be calculated and when some partial product calculations can be omitted. That means the optimal threshold value is important in the iterative MAC performance. To obtain some statistical results on MAC operations, we performed an experiment. We generate normalized 16-bit data x in the range $[-1,1]$. It is hard to simulate with all possible combinations for 16×16 MAC, the result matrix size goes beyond 16 GB. Therefore, we are considering x^2 calculation using the proposed 8-bit iterative MAC unit. The x^2 calculation can show possible minimum and maximum errors with iterative MAC. Three kinds of results are calculated and plotted in Fig. 3.4(a). The blue line shows the output obtained using single iterations $(C' \times D')$, the green line shows the output obtained using three iterations $(C' \times D', C' \times D'', C'' \times D')$ and the red line shows the output obtained using all four iterations $(C' \times D', C' \times D'', C'' \times D', C'' \times D'')$. Note that all outputs were scaled (by 2^8 or 2^{16}) appropriately. As all three output values are very close, the lines are indistinguishable in Fig. 3.4(a). To see the difference, a zoomed version is shown in Fig. 3.4(b). It is observed that the red

line is parabolic as expected as it shows the x^2 function. The blue line shows results for the high byte only, any change in the lower byte value does not reflect the squared value. The squared (blue) value is constant for all adjacent input values whose difference is in the low byte. The green line has a better approximation to x^2 because it includes any change in lower byte values through the additional partial products $(C' \times D'', C'' \times D')$.

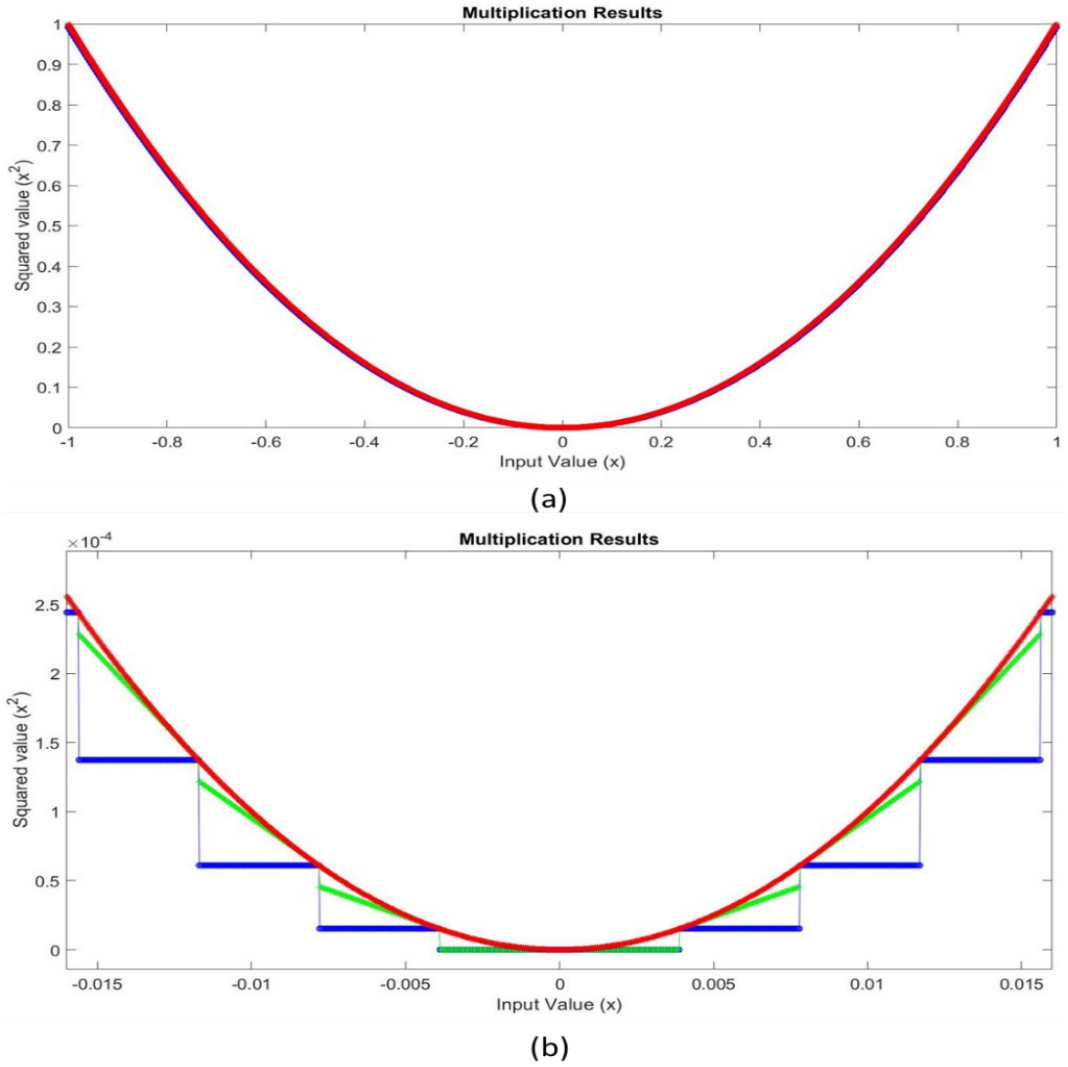


Figure 3.4. 16-bit square value calculation using an 8-bit iterative MAC unit. (a) input vs squared plot of all three results, red indicates full MAC, green indicates three iterations and blue indicates one iteration result. (b) zoom in version of (a) at smaller input values.

The error percentage of the squared values calculated for one (blue) and three (green) iterations are shown in Figures 3.5(a) and (b). Figure 3.5(a) shows the error over the entire range of the input ([-1 1]). It is observed that the percentage error is very small for high values of $|x|$.

A zoomed version of Fig. 3.5(a) at smaller values of $|x|$ is shown in Fig. 3.5(b). Figure 3.5(b) shows that for input values $|x| > 0.15$, the squared values calculated in one iteration are within 5% error (blue line). Note that when we multiply two numbers with uniform probability density function in the range $[-1,1]$, the probability that both operands have a magnitude greater than 0.15 is 72% ($= 0.85 \times 0.85$). In other words, in 72% of the cases, one iteration is sufficient to produce a multiplication result within a 5% error. The error exceeds 5% in the remaining 28% of cases. Note that, as shown in Fig. 3.5(d), the percentage error is high when x becomes small. For example, when $x = 0.05$, the error is around 15%. In other words, when the input is small, the error may be too large to ignore. To obtain a more accurate output, smaller input values require more than one iteration. To simplify the iteration threshold, the size of the value in the first iteration (that is, the high-byte product) is checked. Subsequent iterations are calculated if the value is less than 5% of the maximum possible absolute value. For example, if the first iteration produces 16-bit output then the maximum absolute value in signed 16-bit representation is 32768, and 5% of it is 1639. The threshold value is set to 1639 for an 8-bit iterative MAC, assuming a 5% error can be tolerated in DNNs. If the first iteration value is less than 1,639, the subsequent iterations are performed. The iteration results are scaled by a factor of 2^{16}, which makes the remaining partial products insignificant in the approximated result. The least sixteen bits of the 32-bit register are zeros in a single iteration (blue) calculation. The results can be compressed to a lower size by eliminating zero storage to save memory space. The error in single iteration calculations can be reduced by approximating the least 16 bits with some random data. But the approximation eliminates the possibility of data compression. Therefore, we are not seeking any approximations to reduce the error in partial products. We can choose either input value (based on Fig. 3.4) or partial product value as the threshold for further MAC iterations. We choose a partial product value based (i.e., 1639) as the threshold for further experiments in this work.

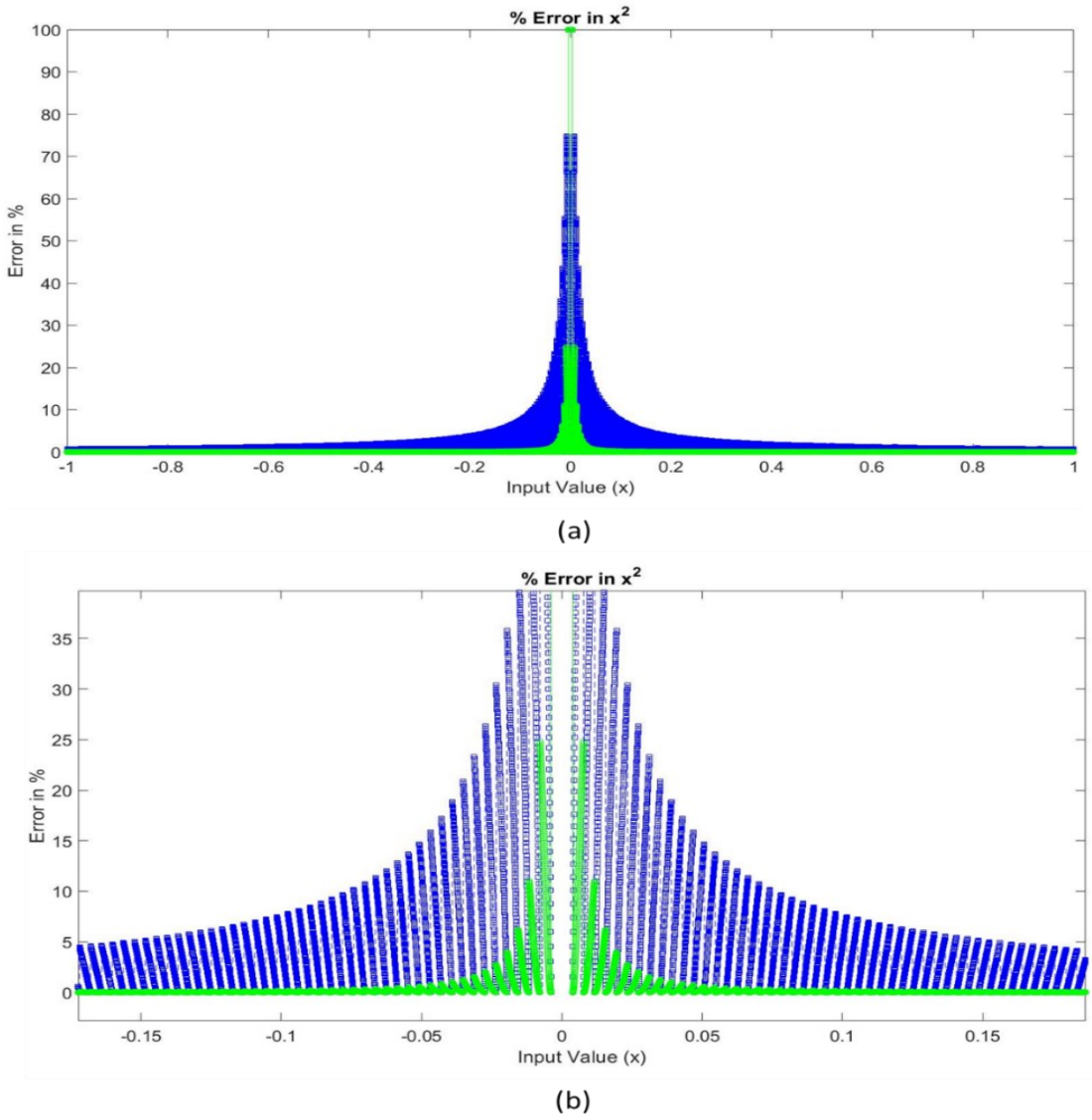


Figure 3.5. 16-bit square value calculation using 8-bit iterative MAC unit (a) percentage of error in square value using single iteration (blue) and three iterations (green) with respect to full precision MAC. (b) zoom in version of (a) at smaller input values.

3.4 Experiments

In this section, we evaluate the performance of the proposed iterative MAC unit for training and testing a popular DNN known as LeNet-300-100 [95]. As shown in Fig. 3.6, LeNet-300-100 is a fully-connected neural network with two hidden layers. The first and second hidden layers contain 300 and 100 neurons, respectively. LeNet-300-100 was initially proposed to classify digit images in the MNIST dataset [99]. The MNIST dataset has 70,000 images, each

28x28 pixels in size, with each image containing a handwritten digit between 0 and 9. The output layer of LeNet-300-100 has ten neurons to classify an input image into one of the ten digits (0 to 9).

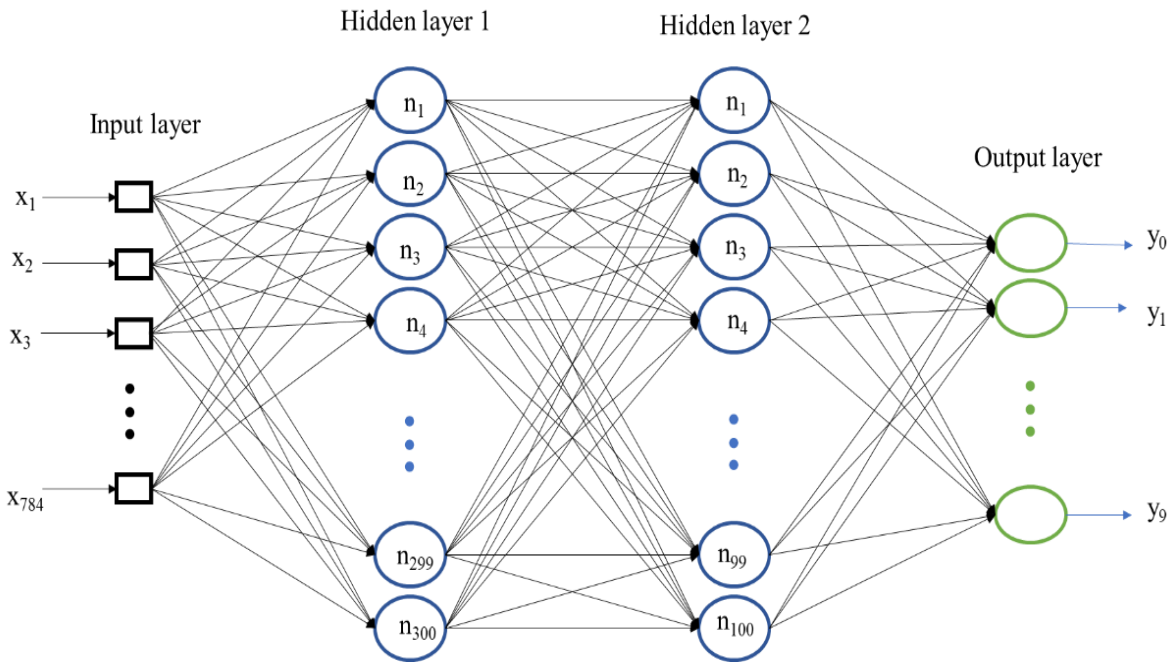


Figure 3.6. LeNet-300-100 network architecture [95].

For the evaluation of the LeNet-300-100 implementation, the MNIST dataset was divided randomly into three sets: training (50,000), validation (10,000), and testing (10,000) images. The training set is used to train the model (i.e., update the model parameters), and the validation set is used to check the performance of the model as the training progresses. The validation data is used to evaluate the model after every iteration. In each iteration, A set of 1000 training images are fed to the network to train the parameters and then a validation set is applied to measure the accuracy with the updated parameters. The validation accuracy typically improves over the iterations, which means parameters move toward the optimal solution. The *validation error* (=1-*validation accuracy*) graph gives an idea about network convergence. After several iterations, the validation accuracy does not improve further, meaning the network converges to an optimal solution/parameters. The optimal parameters might vary based on the weights initialization and training order. Sometimes, the training is stopped before the *validation error* reaches a very small value (zero) to avoid overfitting the model to the training data. The performance of the trained model is evaluated based on unseen test data, and its accuracy is measured.

The LeNet-300-100 was trained and tested with four different precision models, as shown in Table 3.2. Training and testing were simulated in the MATLAB environment. In the first model, the network was trained using single-precision floating-point (FL32) representation for all parameters and data, such as weights, activations, and gradients. This model was used as the reference model to measure the performance of the other models (with fixed-precision representations).

Table 3.2. Four different precision models used for performance evaluation. Note that the Fourth model uses the iterative MAC, whereas the First, Second, and Third models use the regular MAC.

Model	Forward Path	Backpropagation
First model	Weights – FL32 Activations-FL32	Weights – FL32 Local gradient – FL32 Weight gradient – FL32
Second model	Weights – FX8 Activations-FX8	Weights – FL32 Local gradient – FL32 Weight gradient – FL32
Third model	Weights – FX8 Activations-FX8	Weights – FX8 Local gradient – FX40 Weight gradient – FX32
Fourth model	Weights – FX8 Activations-FX8	Weights – FX8 Local gradient – FX40 Weight gradient – FX32

In the second model, the forward propagation parameters (e.g., weights, and activations) are represented in the INT8 format, and the parameters for the backpropagation are in the FL32 format. Note that in the forward path, all arithmetic operations (e.g., multiplications and additions) are done in single-precision floating-point, and the output is quantized to INT8 to use at a later layer. The third model uses a fixed-point format in both forward/inference and backpropagation. A 32-bit accumulator is used in the forward path. Different bit lengths are used at different levels, as shown in Table 3.3. The weights and weight gradients are represented using a 32-bit format, and local gradients are represented using a 40-bit format. The bit lengths used in the third model are not fully optimized as these are the numbers that arise when observing the parameter ranges during the first model training. All computations are performed using the full MAC unit (red line in Fig. 3.4). The fourth model is similar to the third model, but all calculations are performed using an iterative MAC unit with a threshold of 5%. The fourth model replicates the proposed idea of using an iterative MAC unit.

Table 3.3. Parameters' precision in the third and fourth models. Here Partial $\Delta w_{ji}(n) = \delta_j(n) \times x_i(n)$.

Parameter	Fixed-point Bits/ (sign, integer, fraction)
Weights (Forward path)	8 / (1, 0, 7)
Activations (Forward path)	8 / (1, 0, 7)
Local gradient δ_j	40 / (1, 16, 23)
Weight gradient $\Delta w_{ji}(n)$	32 / (1, 0, 31)
Partial weight gradients $\Delta w_{ji}(n)$	32 / (1, 8, 23)
Weights (Stored)	32 / (1, 0, 31)

The convergence curves for all four models are shown in Fig. 3.7. The *validation error* (=1-*validation accuracy*) vs. iteration curve is used to show the convergence during the training. As expected, the first model converged to the best solution. The second model has the largest validation error because it accumulates weights in floating-point and quantizes to INT8 bits for a forward pass. Due to the quantization at each iteration, the weight updates may not reflect the next iteration forward pass, causing the weights to oscillate at local minima. The convergence rate for the third and fourth models is higher than the first, which is suitable for small training datasets but can lead to local minima. The third and fourth models converge at a similar rate and have similar validation and test error, implying that the iterative MAC with a threshold of 5% is as efficient as using a full-size MAC unit.

Table 3.4. Accuracy measured for all four models.

Model	Training accuracy (%)	Testing accuracy (%)
First model	97	96
Second model	74	77
Third model	85	87
Fourth model	86	87

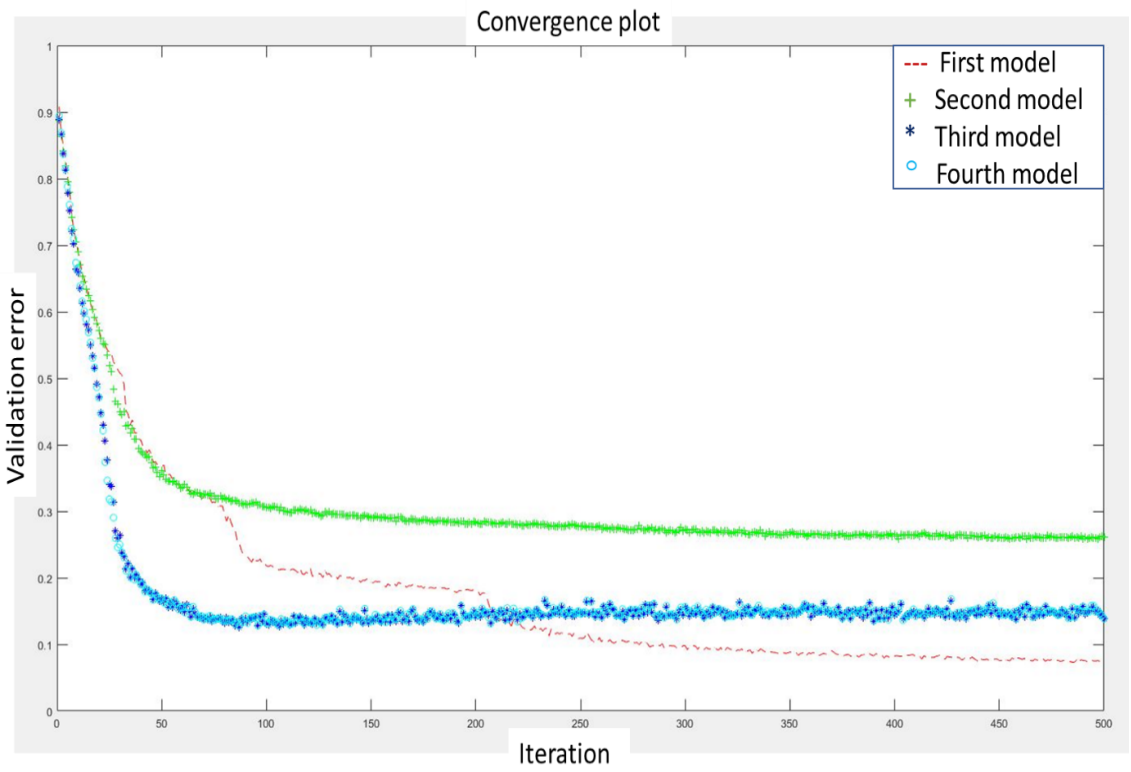


Figure 3.7. Convergence plots of all four models trained.

The trained models are evaluated with 10,000 test images. The results for the Top-1 test and training accuracy for all four models are shown in Table 3.4. As expected, the first model gives the best accuracy. The second model has the lowest accuracy because of the weight quantization at each iteration. The weight updates may not reflect the next iteration forward pass, causing the weights to oscillate at local minima. The third and fourth models have better accuracies than the second and lower accuracy than the first models. The accuracy of the third and fourth models can be further improved with higher bit length in the forward path. The simulation results show that a model can be trained with an iterative MAC.

In the fourth model, during the training, the number of iterations the MAC unit is reused to calculate the local gradient are shown in Fig. 3.8. Figure 3.8 shows the percentage of multiplications requiring the MAC unit to go to a second, third, fourth and fifth iteration (based on 5% threshold value) to calculate 40×8-bit multiplication (local gradient in 40-bit and weight in 8-bit). As the training progresses, the percentage of multiplications requiring the fifth iteration increases due to the decrease in the magnitude of the local gradient. Figure 3.8 shows that the second and third iterations are always required instead of 28% in square value calculation (Fig.

3.4 and 3.5). The local gradient distribution in layers 2 and 3 are plotted in Fig. 3.8 show that the gradient values are not uniformly distributed; most of the gradients are concentrated near zero (very small values). Hence, the second and third iterations are required for almost all the calculations in Lenet-300-100 model. We can also observe that as the gradient distribution is wider in layer 3, the third iteration requirement goes below 90% whereas more than 95% in layer 2. The required iterations might vary depending on the network depth and type.

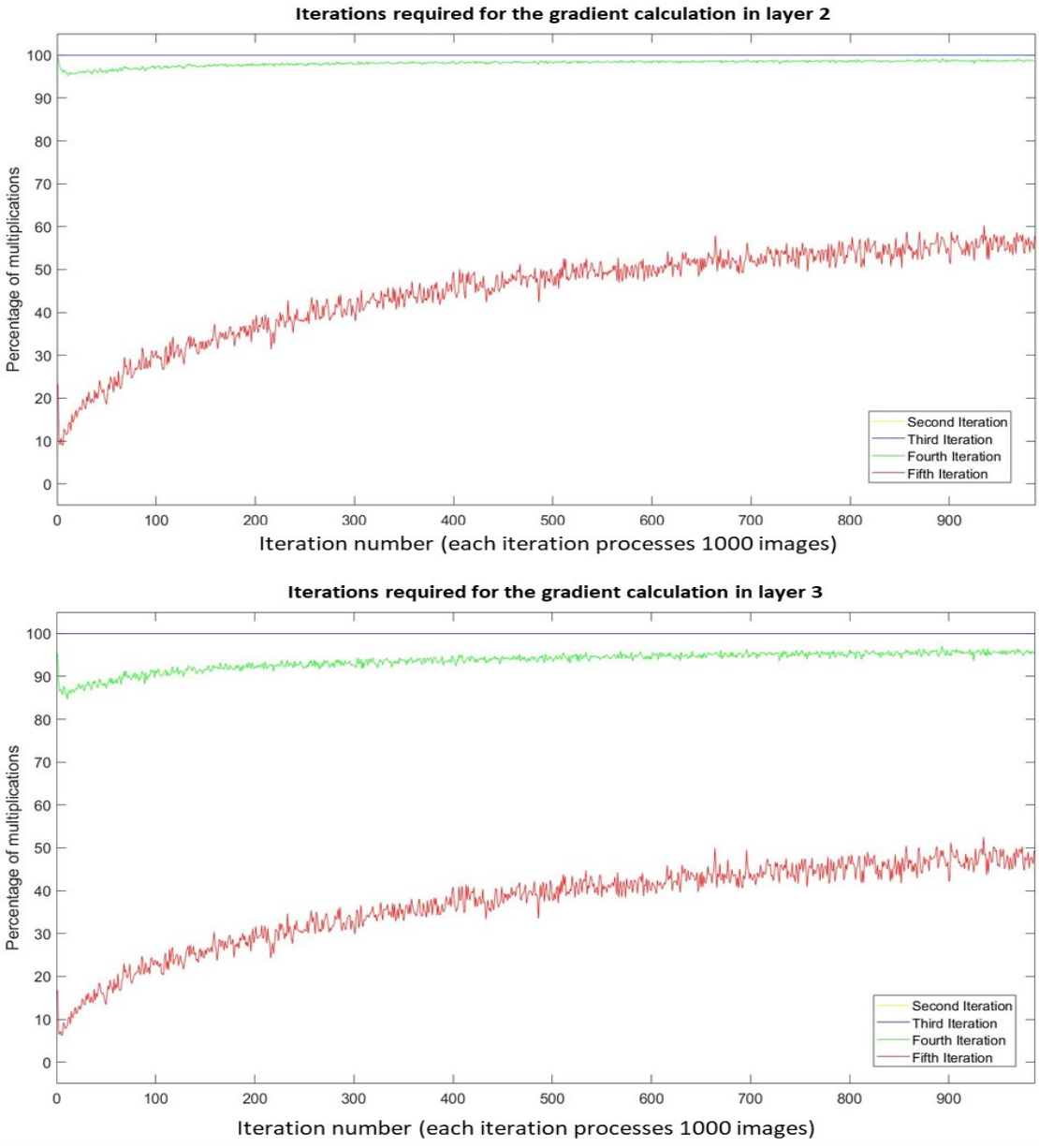


Figure 3.8. Percentage of multiplications in layer 2 and 3 local gradient calculation required second, third, fourth and fifth iterations of iterative MAC unit. The second and third iterations number are similar so only blue line is visible.

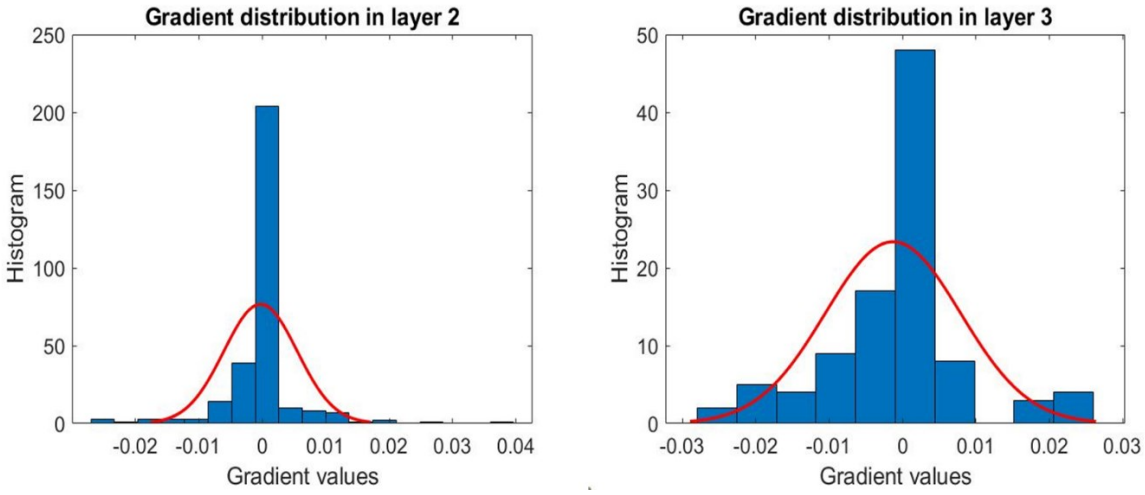


Figure 3.9. Local gradient distribution in layers 2 and 3 of the fourth model.

The simulation results do not generate any insightful information regarding the latency and power reductions. In order to generate those results, the design needs to be implemented on the hardware. For hardware implementation, the existing dataflows discussed in Chapter 2 are ineffective for iterative MAC implementation. In all existing dataflows, full operands (i.e., 8/16/32-bit values) are read from memory for computation, but for an iterative MAC unit, only the upper byte or MSB 8-bits are read first. We may or may not read the next byte value (next 8-bit data) based on the partial product magnitude. Eliminating unnecessary memory accesses for lower-byte data can reduce the memory bandwidth requirement and memory access energy. Therefore, a better dataflow architecture is needed to fully understand the advantages of the iterative MAC at the hardware level.

3.5 Conclusions

In this Chapter, we proposed a low-precision iterative MAC unit-based accelerator for DNN implementation. The proposed accelerator can be used for inference and training the DNN. The popular LeNet-300-100 network was implemented using different precision models, and the performance of the proposed iterative MAC was evaluated. Experimental results show that the iterative MAC is as effective as a full MAC unit because of the error-resilient nature of neural networks.

Chapter 4

Conclusion and Future Work

The DNNs have been shown to deliver state-of-the-art performance in many applications, such as computer vision, medical diagnosis, security, robotics, and autonomous vehicles. The application complexity determines the DNN model size, and large DNN models require more computational power. Therefore, domain-specific hardware accelerators are needed to provide high computational resources with superior energy efficiency and throughput within a small chip area.

In Chapter 2, we identified three major areas: ALU, dataflow, and sparsity, in hardware architectures that can potentially improve an accelerator's overall performance. Existing ASIC hardware accelerators for inference are broadly classified into four categories. Each area offers multiple optimization techniques to improve the overall architecture performance. The advantages and drawbacks of each category are discussed. It is difficult to compare the existing accelerators just based on speed and energy as each accelerator has its own specifications, such as a number of MAC units, on-chip memory size, sparsity in data, and the DNN model. The classification model can help to identify appropriate performance parameters and benchmarks for accelerators. Three major dataflow architectures are evaluated. We found that the dataflow efficiency depends on the workload. Weight-stationary dataflow gives better energy efficiency and row-stationary dataflow has low latency for dense convolutional layers. The performance varies with convolutional stride and filter size. Hence the dataflow must be chosen based on the accelerator application.

DNN deployment on the embedded system requires generality in embedded hardware to produce an optimal performance on DNN models quantized for different precisions. The embedded system should be able to fine-tune the model with the sensor data i.e., training offline. In Chapter 3, we proposed an iterative MAC unit to add precision flexibility and training capabilities to the accelerators. In the iterative MAC, a small MAC unit is reused in time to achieve higher-precision MAC functionality. The number of iterations of the small MAC can be reduced by tracking the error magnitudes. The iterative MAC effectiveness in inference accuracy

and convergence rate at training are measured by simulating the LeNet-300-100 model. The simulation results show that it is as effective as a full MAC unit.

4.1 Future Research Directions

The simulation results of the iterative MAC unit do not give any insights on latency and power reductions. To determine latency and power requirements, we need to implement the DNN model in hardware. For hardware implementation, the existing dataflows discussed in Chapter 2 are ineffective for iterative MAC implementation. In all existing dataflows, full operands (i.e., 8/16/32-bit values) are read from memory for computation, but for an iterative MAC unit, only the upper nibble or MSB 8-bits are read first. We may or may not read the next nibble value (next 8-bit data) based on the partial product magnitude. Eliminating unnecessary memory accesses for lower nibble data can reduce the memory bandwidth requirements and save memory access energy. Therefore, an alternative dataflow which can accesses the upper and lower nibble data independently is needed to fully understand the advantage of the iterative MAC at the hardware level.

References:

1. McCarthy, J. J. (2020). What Is Artificial Intelligence? Artificial Intelligence for Audit, Forensic Accounting, and Valuation, 37–49. https://doi.org/10.1002/9781119601906.ch3
2. Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2017). Imagenet classification with deep convolutional neural networks. Communications of the ACM, 60(6), 84-90. https://doi.org/10.1145/3065386
3. Pierson, H. A., & Gashler, M. S. (2017). Deep learning in robotics: a review of recent research. Advanced Robotics, 31(16), 821–835. https://doi.org/10.1080/01691864.2017.1365009
4. Berman, D. S., Buczak, A. L., Chavis, J. S., & Corbett, C. (2019). A Survey of Deep Learning Methods for Cyber Security. Information, 10(4), 122. https://doi.org/10.3390/info10040122
5. Havaei, M., Davy, A., Warde-Farley, D., Biard, A., Courville, A., Bengio, Y., Pal, C., Jodoin, P., & Larochelle, H. (2016). Brain tumor segmentation with Deep Neural Networks. Medical Image Analysis, 35, 18–31. https://doi.org/10.1016/j.media.2016.05.004
6. Chen, C., Seff, A., Kornhauser, A., & Xiao, J. (2015). Deepdriving: Learning affordance for direct perception in autonomous driving. In Proceedings of the IEEE International Conference on Computer Vision (pp. 2722-2730). https://doi.org/10.1109/ICCV.2015.312
7. Otter, D. W., Medina, J. S., & Kalita, J. (2021). A Survey of the Usages of Deep Learning for Natural Language Processing. IEEE Transactions on Neural Networks and Learning Systems, 32(2), 604–624. https://doi.org/10.1109/tnnls.2020.2979670
8. Akopyan, F., Sawada, J., Cassidy, A., Alvarez-Icaza, R., Arthur, J. M., Merolla, P. A., Imam, N., Nakamura, Y., Datta, P., Nam, G., Taba, B., Beakes, M. P., Brezzo, B., Kuang, J. B., Manohar, R., Risk, W. P., Jackson, B. L., & Modha, D. S. (2015). TrueNorth: Design and Tool Flow of a 65 mW 1 Million Neuron Programmable Neurosynaptic Chip. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 34(10), 1537–1557. https://doi.org/10.1109/tcad.2015.247439
9. NVIDIA Deep Learning Accelerator. (n.d.). http://nvdla.org/index.html
10. Deep-Learning Processor Unit - 3.0 English. (n.d.). https://docs.xilinx.com/r/en-US/ug1414-vitis-ai/Deep-Learning-Processor-Unit
11. Jouppi, N. P., Young, C., Patil, N., Patterson, D. A., Agrawal, G., Bajwa, R., Bates, S. H., Bhatia, S. K., Boden, N., Borchers, A. T., Boyle, R. J., Cantin, P., Chao, C., Clark, C. D., Coriell, J. M., Daley, M. J., Dau, M., Dean, J., Gelb, B., . . . Yoon, D. H. (2017b). In-Datacenter Performance Analysis of a Tensor Processing Unit. Computer Architecture News, 45(2), 1–12. https://doi.org/10.1145/3140659.3080246
12. Albericio, J., Judd, P., Hetherington, T., Aamodt, T., Jerger, N. E., & Moshovos, A. (2016). Cnvlutin: Ineffectual-neuron-free deep neural network computing. ACM SIGARCH Computer Architecture News, 44(3), 1-13. https://doi.org/10.1145/3007787.3001138
13. Judd, P., Lascorz, A. D., Sharify, S., & Moshovos, A. (2017). Cnvlutin2: Ineffectual-Activation-and-Weight-Free Deep Neural Network Computing. ArXiv (Cornell University). http://arxiv.org/pdf/1705.00125.pdf

14. Zhang, S., Du, Z., Zhang, L., Lan, H., Liu, S., Li, L., ... & Chen, Y. (2016, October). Cambricon-X: An accelerator for sparse neural networks. In 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO) (pp. 1-12). IEEE. https://doi.org/10.1109/MICRO.2016.7783723
15. Chen, Y. H., Krishna, T., Emer, J. S., & Sze, V. (2016). Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. IEEE Journal of Solid-State Circuits, 52(1), 127-138. https://doi.org/10.1109/JSSC.2016.2616357
16. Moons, B., Uytterhoeven, R., Dehaene, W., & Verhelst, M. (2017). 14.5 envision: A 0.26-to-10tops/w subword-parallel dynamic-voltage-accuracy-frequency-scalable convolutional neural network processor in 28nm fdsoi. In 2017 IEEE International Solid-State Circuits Conference (ISSCC) (pp. 246-247). IEEE. https://doi.org/10.1109/ISSCC.2017.7870353
17. Han, S., Liu, X., Mao, H., Pu, J., Pedram, A., Horowitz, M. A., & Dally, W. J. (2016). EIE: Efficient inference engine on compressed deep neural network. ACM SIGARCH Computer Architecture News, 44(3), 243-254. https://doi.org/10.1145/3007787.3001163
18. Parashar, A., Rhu, M., Mukkara, A., Puglielli, A., Venkatesan, R., Khailany, B., Emer, J., Keckler, S. W., & Dally, W. J. (2017). SCNN: An accelerator for compressed-sparse convolutional neural networks. ACM SIGARCH Computer Architecture News, 45(2), 27-40. https://arxiv.org/pdf/1708.04485.pdf
19. Shao, Y. S., Clemons, J., Venkatesan, R., Zimmer, B., Fojtik, M., Jiang, N., ... & Keckler, S. W. (2019). Simba: Scaling deep-learning inference with multi-chip-module-based architecture. In Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (pp. 14-27). https://doi.org/10.1145/3352460.3358302
20. Hashemi, S., Anthony, N., Tann, H., Bahar, R. I., & Reda, S. (2017). Understanding the impact of precision quantization on the accuracy and energy of neural networks. Design, Automation, and Test in Europe. https://doi.org/10.23919/date.2017.7927224 diannao
21. Sakr, C., Kim, Y., & Shanbhag, N. R. (2017). Analytical guarantees on numerical precision of deep neural networks. International Conference on Machine Learning, 3007–3016. http://proceedings.mlr.press/v70/sakr17a/sakr17a.pdf
22. Gysel, P., Motamedi, M. H. K., & Ghiasi, S. (2016b). Hardware-Oriented Approximation of Convolutional Neural Networks. ArXiv (Cornell University). https://arxiv.org/pdf/1604.03168.pdf
23. Albanie,. (n.d.). GitHub - albanie/convnet-burden: Memory consumption and FLOP count estimates for convnets. GitHub. https://github.com/albanie/convnet-burden
24. Sze, V., Chen, Y., Yang, T., & Emer, J. (2017). Efficient Processing of Deep Neural Networks: A Tutorial and Survey. Proceedings of the IEEE, 105(12), 2295–2329. https://doi.org/10.1109/jproc.2017.2761740
25. Colangelo, P., Nasiri, N., Nurvitadhi, E., Mishra, A. K., Margala, M., & Nealis, K. (2018). Exploration of Low Numeric Precision Deep Learning Inference Using Intel® FPGAs. Field-Programmable Custom Computing Machines. https://doi.org/10.1109/fccm.2018.00020
26. Talib, M. A., Majzoub, S., Nasir, Q., & Jamal, D. F. (2021). A systematic literature review on hardware implementation of artificial intelligence algorithms. The Journal of Supercomputing, 77(2), 1897–1938. https://doi.org/10.1007/s11227-020-03325-8

27. Guo, K., Zeng, S., Yu, J., Wang, Y., & Yang, H. (2019). [DL] A Survey of FPGA-based Neural Network Inference Accelerators. ACM Transactions on Reconfigurable Technology and Systems, 12(1), 1–26. https://doi.org/10.1145/3289185
28. Li, Z., Wang, Y., Zhi, T., & Chen, T. (2017). A survey of neural network accelerators. Frontiers of Computer Science, 11(5), 746–761. https://doi.org/10.1007/s11704-016-6159-1
29. Chen, Y., Chen, T., Xu, Z., Sun, N., & Temam, O. (2016). DianNao family: energy-efficient hardware accelerators for machine learning. Communications of the ACM, 59(11), 105–112. https://doi.org/10.1145/2996864
30. Camus, V., Mei, L., Enz, C., & Verhelst, M. (2019). Review and Benchmarking of Precision-Scalable Multiply-Accumulate Unit Architectures for Embedded Neural-Network Processing. IEEE Journal on Emerging and Selected Topics in Circuits and Systems, 9(4), 697–711. https://doi.org/10.1109/jetcas.2019.2950386
31. Reuther, A., Michaleas, P., Jones, M. P., Gadepally, V., Samsi, S., & Kepner, J. (2019). Survey and Benchmarking of Machine Learning Accelerators. ArXiv (Cornell University). https://doi.org/10.1109/hpec.2019.8916327
32. Du, Z., Guo, Q., Zhao, Y., Zhi, T., Chen, Y., & Xu, Z. (2020). Self-Aware Neural Network Systems: A Survey and New Perspective. Proceedings of the IEEE, 108(7), 1047–1067. https://doi.org/10.1109/jproc.2020.2977722
33. Chen, Y., Xie, Y., Song, L., Chen, F., & Tang, T. (2020). A Survey of Accelerator Architectures for Deep Neural Networks. Engineering, 6(3), 264–274. https://doi.org/10.1016/j.eng.2020.01.007
34. Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M. S., Berg, A. C., & Fei-Fei, L. (2015). ImageNet Large Scale Visual Recognition Challenge. International Journal of Computer Vision, 115(3), 211–252. https://doi.org/10.1007/s11263-015-0816-y
35. Chen, Q., Xin, C., Zou, C., Wang, X., & Wang, B. (2017). A low bit-width parameter representation method for hardware-oriented convolution neural networks. International Conference on ASIC. https://doi.org/10.1109/asicon.2017.8252433
36. Horowitz, M. (2014). 1.1 Computing's energy problem (and what we can do about it). International Solid-State Circuits Conference. https://doi.org/10.1109/isscc.2014.6757323
37. Hubara, I., Courbariaux, M., Soudry, D., El-Yaniv, R., & Bengio, Y. (2016b). Quantized neural networks: training neural networks with low precision weights and activations. Journal of Machine Learning Research, 18(1), 6869–6898. https://jmlr.org/papers/volume18/16-456/16-456.pdf
38. Jacob, B., Kligys, S., Chen, B., Zhu, M., Tang, M. F., Howard, A. W., Adam, H., & Kalenichenko, D. (2018b). Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference. Computer Vision and Pattern Recognition. https://doi.org/10.1109/cvpr.2018.00286
39. Wu, S., Li, G., Chen, F., & Shi, L. (2018). Training and Inference with Integers in Deep Neural Networks. International Conference on Learning Representations. https://arxiv.org/pdf/1802.04680.pdf
40. Hubara, I., Courbariaux, M., Soudry, D., El-Yaniv, R., & Bengio, Y. (2016). Binarized Neural Networks. Neural Information Processing Systems, 29, 4107–4115. https://papers.nips.cc/paper/6573-binarized-neural-networks.pdf

41. Li, F., & Liu, B. (2016). Ternary Weight Networks. ArXiv (Cornell University). https://doi.org/10.48550/arxiv.1605.04711
42. Judd, P., Albericio, J., Hetherington, T., Aamodt, T. M., Jerger, N. E., Urtasun, R., & Moshovos, A. (2017). Proteus: Exploiting precision variability in deep neural networks. Parallel Computing, 73, 40–51. https://doi.org/10.1016/j.parco.2017.05.003
43. NVIDIA T4 Tensor Core GPUs for Accelerating AI Inference. (n.d.). NVIDIA. https://www.nvidia.com/en-us/data-center/tesla-t4/
44. Wang, E., Davis, J., Zhao, R., Ng, H., Niu, X., Luk, W., Cheung, P. Y. K., & Constantinides, G. A. (2019). Deep Neural Network Approximation for Custom Hardware: Where We've Been, Where We're Going. ACM Computing Surveys. http://arxiv.org/pdf/1901.06955
45. Han, S., Mao, H., & Dally, W. J. (2015d). Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. ArXiv (Cornell University). https://arxiv.org/pdf/1510.00149.pdf
46. Chen, T., Du, Z., Sun, N., Wang, J., Wu, C., Chen, Y., & Temam, O. (2014). Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. Computer Architecture News, 42(1), 269–284. https://doi.org/10.1145/2654822.2541967
47. Chen, Y., Luo, T., Liu, S., Zhang, S., He, L., Wang, J., Li, L., Chen, T., Xu, Z., Sun, N., & Temam, O. (2014). DaDianNao: A Machine-Learning Supercomputer. International Symposium on Microarchitecture. https://doi.org/10.1109/micro.2014.5
48. Du, Z., Fasthuber, R., Chen, T., Ienne, P., Li, L., Luo, T., Feng, X., Chen, Y., & Temam, O. (2016). ShiDianNao: Shifting vision processing closer to the sensor. Computer Architecture News, 43(3S), 92–104. https://doi.org/10.1145/2872887.2750389
49. Daofu, L., Chen, T., Liu, S., Zhou, J., Zhou, S., Teman, O., Feng, X., Zhou, X., & Chen, Y. (2015). PuDianNao: A Polyvalent Machine Learning Accelerator. SIGPLAN Notices, 50(4), 369–381. https://doi.org/10.1145/2775054.2694358
50. Shin, D., Lee, J., & Yoo, H. (2017). 14.2 DNPU: An 8.1TOPS/W reconfigurable CNN-RNN processor for general-purpose deep neural networks. International Solid-State Circuits Conference. https://doi.org/10.1109/isscc.2017.7870350
51. Lee, J., Kim, C., Kang, S., Shin, D., Kim, S., & Yoo, H. (2018). UNPU: A 50.6TOPS/W unified deep neural network accelerator with 1b-to-16b fully-variable weight bit-precision. International Solid-State Circuits Conference. https://doi.org/10.1109/isscc.2018.8310262
52. Kim, E. K., Del Barrio, A. A., Kim, H., & Bagherzadeh, N. (2021). The Effects of Approximate Multiplication on Convolutional Neural Networks. IEEE Transactions on Emerging Topics in Computing, 10(2), 904–916. https://doi.org/10.1109/tetc.2021.3050989
53. Ansari, M., Cockburn, B. F., & Han, J. (2021). An Improved Logarithmic Multiplier for Energy-Efficient Neural Computing. IEEE Transactions on Computers, 70(4), 614–625. https://doi.org/10.1109/tc.2020.2992113
54. Carmichael, Z., Langroudi, H. F., Khazanov, C., Lillie, J. S., Gustafson, J. L., & Kudithipudi, D. (2019). Deep Positron: A Deep Neural Network Using the Posit Number System. Design, Automation, and Test in Europe. https://doi.org/10.23919/date.2019.8715262

55. Olsen, E. (2018). RNS Hardware Matrix Multiplier for High Precision Neural Network Acceleration: "RNS TPU." International Symposium on Circuits and Systems. https://doi.org/10.1109/iscas.2018.8351352
56. Samimi, N., Kamal, M., Afzali-Kusha, A., & Pedram, M. (2020). Res-DNN: A Residue Number System-Based DNN Accelerator Unit. IEEE Transactions on Circuits and Systems I-Regular Papers, 67(2), 658–671. https://doi.org/10.1109/tcsi.2019.2951083
57. Chen, Y., Emer, J., & Sze, V. (2016b). Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks. 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA). https://doi.org/10.1109/isca.2016.40
58. Chen, Y., Yang, T., Emer, J., & Sze, V. (2019). Eyeriss v2: A Flexible Accelerator for Emerging Deep Neural Networks on Mobile Devices. IEEE Journal on Emerging and Selected Topics in Circuits and Systems, 9(2), 292–308. https://doi.org/10.1109/jetcas.2019.2910232
59. Venkatesan, R., Raina, P., Zhang, Y., Zimmer, B., Dally, W. J., Emer, J., Keckler, S. W., Khailany, B., Shao, Y. S., Wang, M., Clemons, J., Dai, S., Fojtik, M., Keller, B., Klinefelter, A., & Pinckney, N. (2019). MAGNet: A Modular Accelerator Generator for Neural Networks. International Conference on Computer Aided Design. https://doi.org/10.1109/iccad45719.2019.8942127
60. Alwani, M., Chen, H. Y. H., Ferdman, M., & Milder, P. (2016). Fused-layer CNN accelerators. 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). https://doi.org/10.1109/micro.2016.7783725
61. Zhang, J., Lee, C., Liu, C., Shao, Y. S., Keckler, S. W., & Zhang, Z. (2019). SNAP: A 1.67 — 21.55TOPS/W Sparse Neural Acceleration Processor for Unstructured Sparse Deep Neural Network Inference in 16nm CMOS. Symposium on VLSI Circuits. https://doi.org/10.23919/vlsic.2019.8778193
62. Lee, J., Lee, J., Han, D., Park, G., & Yoo, H. (2019). 7.7 LNPU: A 25.3TFLOPS/W Sparse Deep-Neural-Network Learning Processor with Fine-Grained Mixed Precision of FP8-FP16. International Solid-State Circuits Conference. https://doi.org/10.1109/isscc.2019.8662302
63. Lin, C., Cheng, C., Tsai, Y., Hung, S., Kuo, Y., Wang, P. G., Tsung, P., Hsu, J., Lai, W., Liu, C., Wang, S., Kuo, C., Chang, C., Lee, M., Lin, T., & Chen, C. (2020). 7.1 A 3.4-to-13.3TOPS/W 3.6TOPS Dual-Core Deep-Learning Accelerator for Versatile AI Applications in 7nm 5G Smartphone SoC. International Solid-State Circuits Conference. https://doi.org/10.1109/isscc19947.2020.9063111
64. Xuda, Z., Du, Z., Guo, Q., Liu, S., Liu, C., Wang, C., Zhou, X., Li, L., Chen, T., & Chen, Y. (2018). Cambricon-S: Addressing Irregularity in Sparse Neural Networks through A Cooperative Software/Hardware Approach. International Symposium on Microarchitecture. https://doi.org/10.1109/micro.2018.00011
65. De Lima, T. F., Peng, H., Tait, A. N., Nahmias, M. A., Miller, H. L., Shastri, B. J., & Prucnal, P. R. (2019). Machine Learning with Neuromorphic Photonics. Journal of Lightwave Technology, 37(5), 1515–1534. https://doi.org/10.1109/jlt.2019.2903474
66. Kim, D., Kung, J., Chai, S. M., Yalamanchili, S., & Mukhopadhyay, S. (2016). Neurocube: A Programmable Digital Neuromorphic Architecture with High-Density 3D Memory. 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA). https://doi.org/10.1109/isca.2016.41

67. Lu, H., Wei, X., Lin, N., Yan, G., & Li, X. (2018). Tetris: re-architecting convolutional neural network computation for machine learning accelerators. Proceedings of the International Conference on Computer-Aided Design. 1-8, https://doi.org/10.1145/3240765.3240855
68. Xiao, T., Bennett, C., Feinberg, B., Agarwal, S., & Marinella, M. J. (2020). Analog architectures for neural network acceleration based on non-volatile memory. Applied Physics Reviews, 7(3), 031301. https://doi.org/10.1063/1.5143815
69. Chi, P., Li, S., Xu, C., Zhang, T., Zhao, J., Liu, Y., Wang, Y., & Xie, Y. (2016). PRIME: A Novel Processing-in-Memory Architecture for Neural Network Computation in ReRAM-Based Main Memory. 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA). https://doi.org/10.1109/isca.2016.13
70. Shafiee, A., Nag, A., Muralimanohar, N., Balasubramonian, R., Strachan, J. P., Hu, M., Williams, R., & Srikumar, V. (2016). ISAAC: A Convolutional Neural Network Accelerator with In-Situ Analog Arithmetic in Crossbars. 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA). https://doi.org/10.1109/isca.2016.12
71. Bojnordi, M. N., & Ipek, E. (2016). Memristive Boltzmann machine: A hardware accelerator for combinatorial optimization and deep learning. High-Performance Computer Architecture. https://doi.org/10.1109/hpca.2016.7446049
72. Nag, A., Balasubramonian, R., Srikumar, V., Walker, R. M., Shafiee, A., Strachan, J. P., & Muralimanohar, N. (2018). Newton: Gravitating Towards the Physical Limits of Crossbar Acceleration. IEEE Micro, 38(5), 41–49. https://doi.org/10.1109/mm.2018.053631140
73. Ankit, A., Hajj, I. E., Chalamalasetti, S. R., Ndu, G., Foltin, M., Williams, R. S., Faraboschi, P., Hwu, W., Strachan, J. P., Roy, K., & Milojicic, D. S. (2019b). PUMA: A programmable ultra-efficient memristor-based accelerator for machine learning inference. InProceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, 715-731. https://doi.org/10.1145/3297858.3304049
74. Yao, P., Wu, H., Gao, B., Tang, J., Zhang, Q., Zhang, W., Yang, J., & Qian, H. (2020). Fully hardware-implemented memristor convolutional neural network. Nature, 577(7792), 641–646. https://doi.org/10.1038/s41586-020-1942-4
75. Cao, N., Chang, M., & Raychowdhury, A. (2019). 14.1 A 65nm 1.1-to-9.1TOPS/W Hybrid-Digital-Mixed-Signal Computing Platform for Accelerating Model-Based and Model-Free Swarm Robotics. International Solid-State Circuits Conference. https://doi.org/10.1109/isscc.2019.8662311
76. Bankman, D., Yang, L., Moons, B., Verhelst, M., & Murmann, B. (2018). An Always-On 3.8uJ/86% CIFAR-10 Mixed-Signal Binary CNN Processor with All Memory on Chip in 28-nm CMOS. IEEE Journal of Solid-State Circuits, 54(1), 158–172. https://doi.org/10.1109/jssc.2018.2869150
77. Tsai, H., Ambrogio, S., Narayanan, P., Shelby, R. M., & Burr, G. W. (2018). Recent progress in analog memory-based accelerators for deep learning. Journal of Physics D, 51(28), 283001. https://doi.org/10.1088/1361-6463/aac8a5
78. Parashar, A., Raina, P., Shao, Y. S., Chen, Y., Ying, V. A., Mukkara, A., Venkatesan, R., Khailany, B., Keckler, S. W., & Emer, J. (2019). Timeloop: A Systematic Approach to

DNN Accelerator Evaluation. International Symposium on Performance Analysis of Systems and Software. https://doi.org/10.1109/ispass.2019.00042
79. Yang, T., Chen, Y., Emer, J., & Sze, V. (2017). A method to estimate the energy consumption of deep neural networks. Asilomar Conference on Signals, Systems and Computers. https://doi.org/10.1109/acssc.2017.8335698
80. K. Guo, W. Li, K. Zhong, Z. Zhu, S. Zeng, S. Han, Y. Xie, P. Debacker, M. Verhelst, Y. Wang. Neural Network Accelerator Comparison, [Online] https://nicsefc.ee.tsinghua.edu.cn/projects/neural-network-accelerator/
81. Lai, L., Suda, N., & Chandra, V. (2017). Deep Convolutional Neural Network Inference with Floating-point Weights and Fixed-point Activations. ArXiv (Cornell University). https://arxiv.org/pdf/1703.03073.pdf
82. Jacob, B., Kligys, S., Chen, B., Zhu, M., Tang, M. F., Howard, A. W., Adam, H., & Kalenichenko, D. (2018). Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference. Computer Vision and Pattern Recognition. https://doi.org/10.1109/cvpr.2018.00286
83. Ma, Y., Suda, N., Cao, Y., Seo, J., & Vrudhula, S. (2016). Scalable and modularized RTL compilation of Convolutional Neural Networks onto FPGA. Field-Programmable Logic and Applications. https://doi.org/10.1109/fpl.2016.7577356
84. Courbariaux, M., Bengio, Y., & David, J. (2015). BinaryConnect: training deep neural networks with binary weights during propagations. Neural Information Processing Systems, 28, 3123–3131. https://arxiv.org/pdf/1511.00363
85. Zhu, C., Han, S., Mao, H., & Dally, W. J. (2016). Trained Ternary Quantization. ArXiv (Cornell University). https://doi.org/10.48550/arxiv.1612.01064
86. Rastegari, M., Ordonez, V., Redmon, J., & Farhadi, A. (2016). XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks. Lecture Notes in Computer Science, 525–542. https://doi.org/10.1007/978-3-319-46493-0_32
87. Courbariaux, M., Hubara, I., Soudry, D., El-Yaniv, R., & Bengio, Y. (2016). Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1. ArXiv (Cornell University). https://arxiv.org/pdf/1602.02830.pdf
88. Zhou, S., Wu, Y., Zekun, N., Zhou, X., Wen, H., & Zou, Y. (2016). DoReFa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients. ArXiv (Cornell University). https://doi.org/10.48550/arxiv.1606.06160
89. Cai, Z., He, X., Sun, J., & Vasconcelos, N. (2017). Deep Learning with Low Precision by Half-Wave Gaussian Quantization. Computer Vision and Pattern Recognition. https://doi.org/10.1109/cvpr.2017.574
90. Miyashita, D., Lee, E. A., & Murmann, B. (2016). Convolutional Neural Networks using Logarithmic Data Representation. ArXiv (Cornell University). http://export.arxiv.org/pdf/1603.01025
91. Zhou, A., Yao, A., Guo, Y., Xu, L., & Chen, Y. (2017). Incremental Network Quantization: Towards Lossless CNNs with Low-Precision Weights. ArXiv (Cornell University). https://arxiv.org/pdf/1702.03044.pdf
92. Micikevicius, P., Narang, S., Alben, J. M., Diamos, G., Elsen, E., Garcia, D. A., Ginsburg, B., Houston, M. J., Kuchaiev, O., Venkatesh, G., & Wu, H. (2017). Mixed Precision Training. ArXiv (Cornell University). https://doi.org/10.48550/arxiv.1710.03740

93. Wang, N., Choi, J., Brand, D., Chen, C., & Gopalakrishnan, K. (2018). Training Deep Neural Networks with 8-bit Floating Point Numbers. Neural Information Processing Systems, 31, 7675–7684. https://papers.nips.cc/paper/2018/file/335d3d1cd7ef05ec77714a215134914c-Paper.pdf.
94. Das, D., Mellempudi, N., Mudigere, D., Kalamkar, D. D., Avancha, S., Banerjee, K., Sridharan, S., Vaidyanathan, K., Kaul, B., Georganas, E., Heinecke, A., Dubey, P., Corbal, J., Shustrov, N., Dubtsov, R. S., Fomenko, E., & Pirogov, V. O. (2018). Mixed Precision Training of Convolutional Neural Networks using Integer Operations. ArXiv (Cornell University). https://arxiv.org/pdf/1802.00930
95. Deng, L. (2012). The MNIST Database of Handwritten Digit Images for Machine Learning Research [Best of the Web]. IEEE Signal Processing Magazine, 29(6), 141–142. https://doi.org/10.1109/msp.2012.221147
96. Shawahna, A., Sait, S. M., & El-Maleh, A. (2018). FPGA-based accelerators of deep learning networks for learning and classification: A review. IEEE Access, 7, 7823-7859.
97. Ruder, S. (2016). An overview of gradient descent optimization algorithms. arXiv preprint arXiv:1609.04747.
98. Machupalli, R., Hossain, M., & Mandal, M. (2022). Review of ASIC accelerators for deep neural network. Microprocessors and Microsystems, 89, 104441. https://doi.org/10.1016/j.micpro.2022.104441
99. Deng, L. (2012). The mnist database of handwritten digit images for machine learning research. IEEE Signal Processing Magazine, 29(6), 141–142.