August 2023
Master's Degree Thesis

Hardware Acceleration of Fused-Layer Convolutional Neural Networks via Most-Significant-Digit First Arithmetic

Graduate School of Chosun University

Department of Computer Engineering

Mohammadhosein Gholamrezaei

Hardware Acceleration of Fused-Layer Convolutional Neural Networks via Most-Significant-Digit First Arithmetic

MSDF (Most-Significant-Digit First Arithmetic) 연산기법을 적용한 Fused-layer CNN 하드웨어 가속기 연구

August 25, 2023

Graduate School of Chosun University

Department of Computer Engineering

Mohammadhosein Gholamrezaei

Hardware Acceleration of Fused-Layer Convolutional Neural Networks via Most-Significant-Digit First Arithmetic

Advisor: Prof. Jeong-A, Lee

This Thesis is submitted to Graduate School of Chosun University in partial fulfillment of the requirements for a Master's degree

April 2023

Graduate School of Chosun University

Department of Computer Engineering

Mohammadhosein Gholamrezaei

골람레자이 모함마도세인의
석사학위논문을 인준함

위원장 조선대학교 교 수 신석주 (인)
위 원 조선대학교 교 수 강문수 (인)
위 원 조선대학교 교 수 이정아 (인)

2023년 5월

조선대학교 대학원

Contents

한 글 요 약

MSDF (Most-Significant-Digit First Arithmetic) 연산기법을 적용한
Fused-layer CNN 하드웨어 가속기 연구

골람레자이 모함마도세인
지도교수: 이정아
컴퓨터공학과
조선대학교 대학원

DNN(심층 신경망)은 데이터 내에서 복잡한 패턴과 관계를 학습할 수 있는 기능으로 인해 최근 몇 년 동안 점점 인기를 얻고 있습니다. 합성곱 신경망(CNN)은 복잡한 이미지 및 비디오 데이터를 효과적으로 분석하고 분류하는 기능으로 인해 최근 몇 년 동안 인기를 얻은 일종의 심층 신경망입니다.

융합 계층 데이터 흐름은 네트워크 내에서 데이터 처리를 최적화하기 위해 CNN에서 사용되는 기술입니다. 여러 계층을 단일 작업으로 결합하면 계층 간에 전송해야 하는 데이터 양이 줄어들어 네트워크의 계산 효율성과 메모리 요구 사항이 모두 크게 향상될 수 있습니다.

MSDF(Most-Significant-Digit First) 산술은 자릿수 수준의 파이프라이닝과 고유한 가변 정밀도 기능이 있어 CNN을 위한 유망한 산술 기술입니다. 기존의 비트 병렬 및 비트 직렬 산술과 달리 MSDF는 다음 종속 작업으로 이동하기 전에 작업이 완료될 때까지 기다리지 않고 연결을 통해 연속 작업의 중첩 실행을 허용합니다. ReLU 계층에서는 음수 출력이 감지되는 즉시 계산을 종료할 수 있는 반면 MaxPool 계층에서는 최대값이 감지되는 즉시 종료될 수 있습니다.

이 논문에서는 MSDF 산술을 사용하고 연속 레이어에 걸쳐 숫자 수준 파이프라이닝을 가능하게 하는 동시에 비효율적인 계산을 조기에 종료하는 CNN용 하드웨어 가속기를 제안합니다. CNN에서 연속 레이어를 융합하면 MSDF 산술에서 조기 종료 비율을 높일 수 있습니다. 우리의 평가는 LeNet-5의 처음 두 계층이 함께 융합될 때 최대 58.2%의 계산을 건너뛸 수 있음을 보여줍니다.

Abstract

Hardware Acceleration of Fused-Layer Convolutional Neural Networks via Most-Significant-Digit First Arithmetic

Mohammadhosein Gholamrezaei
Advisor: Prof. Jeong-A, Lee
Department of Computer Engineering
Graduate School of Chosun University

Deep neural networks (DNNs) have become increasingly popular in recent years due to their ability to learn complex patterns and relationships within data. Convolutional neural networks (CNNs) are a type of deep neural network that have gained popularity in recent years due to their ability to effectively analyze and classify the complex image and video data.

Fused-layer dataflow is a technique used in CNNs to optimize the processing of data within the network. By fusing multiple layers together into a single operation, the amount of data that needs to be transferred between layers is reduced, which can lead to significant improvements in both the computational efficiency and memory requirements of the network.

Most-Significant-Digit First (MSDF) arithmetic, with its digit-level pipelining and inherent variable precision capabilities, is a promising arithmetic technique for CNNs. Unlike traditional bit parallel and bit serial arithmetic, MSDF allows for overlapping execution of successive operations through chaining, rather than waiting for an operation to finish before moving on to the next dependent operation. In ReLU layers, the computation can be terminated as soon as a negative output is detected, while in MaxPool layers, it can be terminated as soon as the maximum is detected.

In this thesis, we propose a hardware accelerator for CNNs that uses MSDF arithmetic and enables digit-level pipelining across successive layers, while also terminating ineffective computations early. By fusing successive layers in CNNs,

the rate of early termination in MSDF arithmetic can be increased. Our evaluations demonstrate that up to 56.3% of computations can be skipped when the first two layers of LeNet-5 are fused together.

Index Terms: Convolutional Neural Networks, Fused-layer Dataflow, Most-Significant-Digit First Arithmetic, Run-time Pruning, Early Termination, Hardware Accelerator.

Acronyms

ASIC	Application-Specific Integrated Chip
BSD	Binary Signed-Digit
CA	Conversion/Append
CMOS	Complementary Metal-Oxide-Semiconductor
CNN	Convolutional Neural Network
CPU	Central Processing Unit
CS	Carry-Save
CV	Computer Vision
DNN	Deep Neural Network
DRAM	Dynamic Random-Access Memory
FA	FA
FPGAs	Field Programmable Gate Array
FSM	Finite-State Machine
GPU	Graphics Processing Unit
HA	Half Adder
IEN	Inverted Encoding for Negabit
LR	Left-to-Right
LSD	Least Significant Digit
MAC	Multiply-accumulate
MSD	Most Significant Digit
MSDF	Most Significant Digit First
NoC	Network-On-Chip
PE	Processing Element
PIM	Processing-in-Memory
PPR	Partial Product Row
ReLU	Rectified Linear Unit
SD	Signed-digit
SIMD	Single-Instruction Multiple-Data
SIMT	Single-Instruction Multiple-Thread
TPU	Tensor Processing Unit

List of Figures

List of Tables

Chapter I.
Introduction

A. Research Motivation

Deep Neural Networks (DNNs) have garnered considerable attention in recent times owing to their impressive achievements across a wide range of applications, including image and speech recognition, natural language processing, and autonomous vehicles, among others [1]. Convolutional Neural Networks (CNNs) are a specific type of DNNs widely used in image and video processing tasks. They have a hierarchical structure that learns complex features from raw data and performs convolution operations to extract spatial information [2].

Currently, in Computer Vision (CV) systems, CNNs are considered the most promising method for image understanding. These algorithms are inspired by the human brain and are composed of several layers of feature detectors and classifiers, which are optimized using machine learning techniques [3]. Although the concept of neural networks has existed for almost 80 years [4], it is only with the latest high-performance computing hardware that it has become possible to effectively evaluate and train CNNs with sufficient depth and breadth to achieve good performance in image understanding applications. However, in recent years, progress has been remarkable, and the state-of-the-art CNNs can now compete with human accuracy in image classification tasks [5].

However, CNN inference on large datasets and real-time applications can pose significant challenges due to their computational complexity and memory requirements. CNN inference involves a large number of multiply-accumulate (MAC) operations that consume a significant amount of power and time. Additionally, the large size of CNN models requires a significant amount of memory, which can lead to memory access and storage challenges. Figure 1. demonstrates popular CNN architectures required MAC operations and their model size [6].

Therefore, there is a growing demand for hardware acceleration techniques that can efficiently execute CNNs on specialized hardware platforms. These techniques aim to reduce the computational time and energy consumption of CNN inference while maintaining the accuracy of the results.

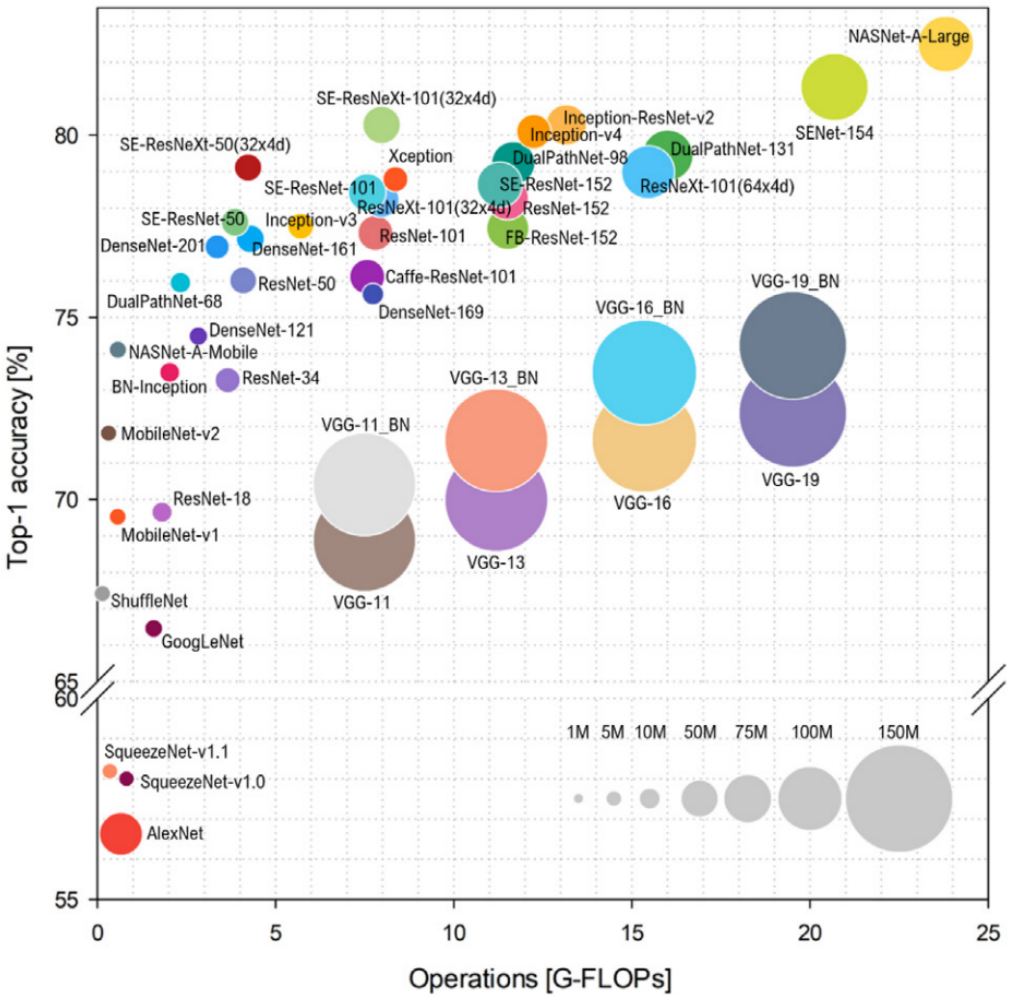


Figure 1: Popular CNN Models: Top-1 accuracy vs GFLOPs.

Implementing CNNs on hardware accelerators poses several challenges that need to be addressed. One of the most significant challenges is memory access, as CNNs require a lot of memory to store weights, activations, and feature maps. This can lead to memory bandwidth and storage challenges. Another challenge is the computational intensity of CNNs, which involve a large number of MAC operations that consume a significant amount of power and time. Hardware accelerators need to be designed to perform these operations efficiently. Additionally, hardware accelerators need to be scalable to the number of layers and size of the CNN model while maintaining accuracy and flexibility to adapt to changes in CNN architectures and parameters. Addressing these challenges is crucial for developing efficient and accurate hardware accelerators for CNNs to meet the demands of real-time applications.

B. Research Objectives

CNN hardware accelerators employ various optimization techniques to improve performance and efficiency, such as weight pruning, quantization, data compression, and dataflow optimation.

Fused-layer dataflow is a technique used in CNNs to optimize their computational efficiency [7]. It involves merging multiple layers of a CNN into a single fused layer so that the output of one layer is directly passed as input to the next layer without intermediate storage. In the fused-layer dataflow, the convolution operation is fused with the activation function and other operations such as batch normalization, resulting in a single fused layer. In fused-layer dataflow, even different convolutional layers can be fused. This reduces the number of memory accesses required during the computation, as intermediate results are not stored in memory.

Quantization is a technique that can improve the performance of CNN accelerators by reducing the precision of the weights and activations from 32-bit floating point to 8-bit or even lower. This reduces the memory bandwidth and storage requirements, allowing more data to fit into on-chip memory and reducing off-chip memory accesses.

To employ the opportunities (i.e., variable bit precision computations) provided by quantization, Most-Significant-Digit First (MSDF) arithmetic is a potential arithmetic technique that uses digit-level pipelining [8]. In contrast to conventional bit parallel and bit serial arithmetic, MSDF allows for the overlapping execution of successive operations through chaining, eliminating the need to wait for a dependent operation to finish. This technique allows for early termination of computations in ReLU layers as soon as a negative output is detected, and in MaxPool layers as soon as the maximum value is detected.

To increase the rate of early termination in MSDF arithmetic, fused-layer dataflow can play an important role and reduce memory bandwidth and improving the energy efficiency of CNN hardware accelerators.

C. Contributions

In this research, a hardware accelerator for CNNs is proposed that leverages MSDF arithmetic for fusing successive layers of CNNs to terminate ineffective computations introduced by ReLU and MaxPool. The main contributions of this thesis are as follows:

- A MaxPool unit has been proposed, and as far as our knowledge goes, no previous proposals for early termination opportunities in the MaxPool layer have been made.
- A Processing Element (PE) has been designed and implemented for the proposed technique.
- An analysis of the potential speedup and memory bandwidth savings has been conducted using an in-house emulator.
- The area, power, and throughput of the proposed design have been evaluated, and a comparison with state-of-the-art Online arithmetic hardware accelerators has been conducted.

D. Thesis Organization

This thesis is structured as follows. Chapter II. provides an overview of the topic and a review of previous research. Chapter III. describes the proposed early termination schemes. Chapter IV. describes the proposed multiplication and inner product unit architecture. Finally, Chapter VI. concludes the research and outlines potential future enhancements.

Chapter II.
Background

A. Deep Neural Networks

DNNs in artificial intelligence attempt to mimic this biological system by using nodes (or artificial neurons) and connections (or artificial synapses) to perform computations [9]. These nodes receive input signals, which are then weighted and combined before being passed on to other nodes in the network. This process continues until an output signal is produced.

Neural networks possess a significant benefit in their capacity to acquire knowledge and accommodate fresh information. This is accomplished via a training procedure wherein the weights of node connections are modified according to input-output pairs. Through successive weight adjustments, the network becomes proficient in identifying patterns and making predictions using novel input data [10].

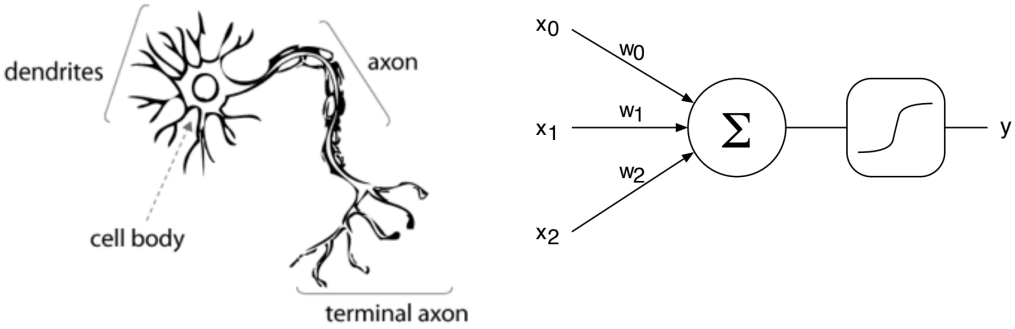


Figure 2: Showing a comparison between a biological neuron and its artificial counterpart (adapted from [11]).

Artificial neurons serve as the fundamental components of artificial neural networks (depicted in Figure 2). They are modeled after the biological neurons in the human brain, which receive input signals from other neurons, process the information, and produce output signals [12].

The artificial neuron receives a set of input signals that are multiplied by respective weights. The weights govern the magnitude of the connection between the input signal and the neuron. The weighed input signals are then summed up and passed through a non-linear activation function. The activation function de-

termines whether the neuron should fire or not based on the weighted inputs. If the input signal is strong enough, the neuron will fire, producing an output signal. The weights of the artificial neuron can be adjusted during the training process to optimize the neuron's response to certain inputs. The functionality of an artificial neuron can be represented by Equation (A..1), where the activation function is denoted by F, bias by b, and N represents the number of input activations.

$$y = F(\sum_{i=1}^{N} x_i w_i + b) \tag{A..1}$$

The organization of a neural network typically involves arranging artificial neurons into layers and connecting them in a specific way. Within a feed-forward neural network architecture, the neurons are organized in a directed acyclic graph, permitting connections solely between neurons in contiguous layers. The neurons are organized into layers according to their location within the network. The initial layer, known as the input layer, receives the input data into the network. The output layer is the final layer in the network, where the output signals are produced. Any layers between the input and output layers are referred to as hidden layers, as they do not directly interact with the input or output data.

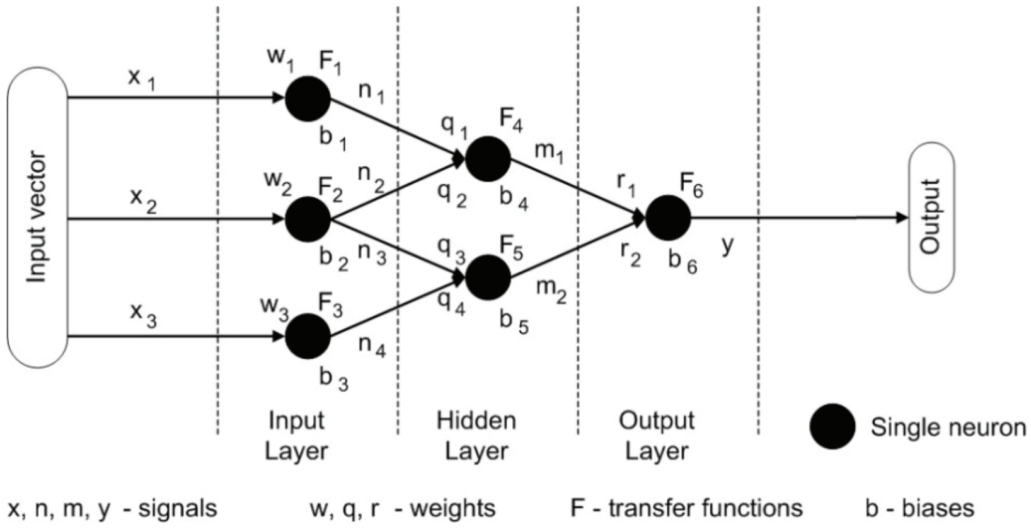


Figure 3: Feed-forward artificial neural network (adapted from [12]).

In a fully connected layer, each neuron in one layer is connected to every neuron in the adjacent layer. This allows for a high degree of connectivity between

the neurons and enables the network to process complex data sets. The number of layers and neurons in each layer can vary depending on the specific task the network is designed to perform. In general, deeper neural networks with more layers tend to perform better on complex tasks, but may require more computational resources and longer training times. Figure 3 illustrates a basic feed-forward artificial neural network consisting of multiple layers, which is used for analytical description and described by sets of equations (A..2), and (A..3).

$$\begin{aligned} n_1 &= F_1(w_1 x_1 + b_1) \\ n_2 &= F_2(w_2 x_2 + b_2) \\ n_3 &= F_2(w_2 x_2 + b_2) \\ n_4 &= F_3(w_3 x_3 + b_3) \end{aligned} \tag{A..2}$$

$$\begin{aligned} m_1 &= F_4(q_1 x_1 + q_2 x_2 + b_4) \\ m_2 &= F_5(q_3 x_3 + q_4 x_4 + b_5) \\ y &= F_6(r_1 m_1 + r_2 m_2 + b_6) \end{aligned} \tag{A..3}$$

B. Convolutional Neural Networks

CNNs are a type of deep learning algorithm that are specifically designed to work with images and other multidimensional data. CNNs are inspired by the organization of the visual cortex in animals, where neurons are arranged in a hierarchical manner and respond to increasingly complex visual patterns. CNNs have become a popular tool for image recognition, object detection, and many other applications in CV.

The key difference between CNN and DNNs is that CNNs are specifically designed for image and other multidimensional data, whereas DNNs can work with any type of data. DNNs consist of multiple layers of neurons that are fully connected, meaning that each neuron in one layer is connected to every neuron in the next layer. In contrast, CNNs consist of multiple layers of neurons that are

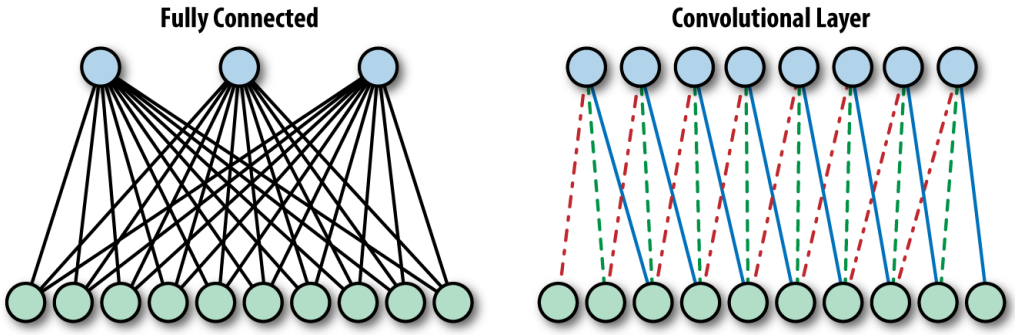


Figure 4: Fully connected layers vs. Convolutional layers (adapted from [13]).

only partially connected, with each neuron in a layer only connected to a small subset of neurons in the next layer. Figure 4 demonstrates the difference between the convolutional layer and the fully connected layer. This is accomplished by employing convolutional layers that employ a collection of filters to process the input image and extract distinctive features.

CNNs excel in image processing due to their ability to leverage local correlations within image data, enabling them to learn effectively from limited training examples. This is made possible by employing convolutional layers that employ a series of filters to extract distinctive characteristics from the input image. These filters are specifically designed to detect particular patterns like edges, corners, and textures present in the image data. By utilizing these filters across the entire image, CNNs significantly decrease the number of parameters that necessitate learning, thereby surpassing fully connected DNNs in efficiency.

The basic components of a CNN are as follows:

Convolutional Layers: Within these layers, a collection of adjustable filters is implemented to process the input image, resulting in a set of activation maps. Each filter operates on a specific local region of the input image, generating a scalar value that quantifies the resemblance between the filter and the given local region.

Pooling Layers: These layers perform downsampling on the activation maps generated by the convolutional layers. By employing operations like maximum or average pooling on a region of activations, they effectively decrease the spatial dimensions of the activation maps. This reduction in dimensionality serves to

minimize the number of parameters in the network, enhancing computational efficiency, while simultaneously preserving crucial spatial details.

Activation Function: At the output of every neuron in the network, an activation function is employed to convert the input into a nonlinear output. Typical activation functions employed in CNNs encompass Rectified Linear Unit (ReLU), sigmoid, and tanh.

Fully Connected Layers: After receiving the output from the convolutional and pooling layers, these layers transform it into a one-dimensional vector by flattening the data. Subsequently, a set of weights is applied to this vector to generate the ultimate output of the network. Fully connected layers are typically employed in the concluding stages of the network, as they are responsible for mapping the learned high-level features from the preceding layers to precise outputs or classifications.

Dropout: A regularization technique applied to reduce overfitting in deep learning networks, dropout randomly "turns off" some neurons in the network during training to force the remaining neurons to learn more robust features.

Batch Normalization: A technique applied to normalize the output of each layer to accelerate training, improve generalization and reduce overfitting.

C. Deep Neural Network Hardware Accelerators

In recent years, there has been a significant surge in the progress of DNN solutions across a wide range of applications. This progress has been made possible by the introduction of fast Graphic Processing Units (GPUs), which effectively handle the increasing memory requirements and computational complexity resulting from the growing size of DNNs. As a result, the deployment of DNNs on edge devices with constrained hardware resources and limited energy availability has become increasingly appealing, expanding the reach of deep learning solutions beyond high-performance computing machines. The hardware options for developing and deploying DNNs encompass general-purpose architectures such as Central Processing Units (CPUs) and GPUs, as well as spatial architectures like Field Programmabel Gate Arrays (FPGAs) and Application-Specific Inte-

grated Chips (ASICs).

The MAC operation plays a fundamental role in both the fully connected and convolutional layers of DNNs. It can be parallelized to achieve rapid inference speeds in both of these layer types. Hardware accelerators can take the form of conventional hardware optimizations that improve compute parallelism, or they can be contemporary accelerators that integrate hardware and software design capabilities. The latest progress in software-based DNN solutions have also demonstrated promising performance improvements by reducing memory usage and computational operations. The prevailing direction in the development of efficient DNN applications involves hardware-software co-design for DNN acceleration.

Subsequently, we will delve into the subsequent sections to review the suitability of various temporal and spatial hardware architectures for DNNs. Furthermore, our emphasis will be on the co-design of DNN accelerators that harness the advantages of both compression algorithms and hardware architecture.

1. Temporal and Spatial Hardware Architectures

Temporal and spatial hardware architectures are the two main types of architectures used in DNN accelerators. Temporal architectures, such as CPUs and GPUs, utilize a large number of ALUs without local memory, while spatial architectures, like ASICs and FPGAs, employ Processing Elements (PEs) with their own local memory and control logic.

In terms of parallelism, CPUs use the Single-Instruction Multiple-Data (SIMD) model, while GPUs use the Single-Instruction Multiple-Thread (SIMT) execution model. Temporal architectures map DNN layers, like fully connected and convolution layers, to matrix multiplication operations. Software libraries such as OpenBLAS and cuBLAS are commonly used for optimizing matrix multiplications on CPUs and GPUs, respectively. However, these architectures are not specifically designed for DNN applications.

DNN accelerators, implemented on ASICs or FPGAs, face memory access as a bottleneck. These architectures utilize an array of PEs that incorporate local and global buffers, thereby minimizing data access from the DRAM. The PE array op-

erates as a two-dimensional Network-On-Chip (NoC) where dataflow processing takes place, allowing direct message passing between PEs. Different DNN accelerator designs can exploit data reuse, such as convolution, filter weight, and input feature map, by leveraging local memory hierarchy. These designs can be classified into four categories: row-stationary, output-stationary, input-stationary, and weight-stationary. Each category utilizes different dataflow characteristics to optimize computation and memory access.

2. Co-Design of Hardware Architecture and Compression Algorithm

CNN compression algorithms, such as pruning and quantization, are commonly employed to achieve efficient CNN structures. To further optimize hardware acceleration for compressed CNN structures, specialized hardware accelerators can be designed. Unstructured pruning removes unimportant weights or activations, significantly reducing off-chip memory access [14–16]. Quantization operates with low-bit precision, reducing computation and memory storage [17–20]. Hardware-aware NAS techniques aim to automatically enhance CNN performance on specific hardware targets, considering factors like inference latency, energy consumption, and memory usage [21–24].

Pruned CNNs often exhibit a high level of sparsity, where many weights or connections can be pruned to zero without significant accuracy loss. Sparse hardware accelerators can be designed to leverage this sparsity, resulting in energy and storage savings. Examples include Cambricon-X [25], which skips MAC operations for zero weights and accesses required weights using index-based storage. Cambricon-S [26] improves weight indexing overhead by employing a cooperative hardware-software approach. SCNN [27] supports convolutional layer processing in a compressed domain, utilizing zero-valued weights and activations. Eyeriss-v2 [28] utilizes the Eyeriss-like row stationary dataflow [29] to process nonzero weights and activations in a compressed domain, reducing memory access overhead. SNAP [30] employs associative index-matching search to find matching non-zero input activations and weight kernels, supporting various layer types with reduced write-back traffic and memory access through two-level par-

tial sum reduction.

These hardware accelerators optimize the performance of compressed CNNs, providing energy efficiency and memory savings through techniques like sparsity exploitation and compressed domain processing.

Quantized architectures aggressively reduce the bit width of weights and activations, even down to 1-bit, to achieve ultra-high inference speeds at the cost of some accuracy loss. Specialized hardware accelerators have been developed to support quantized neural networks using both variable-bitwidth arithmetic (e.g., Stripes [31], BitFusion [32], UNPU [33], BitBlade [34]) and fixed-bitwidth arithmetic (e.g., YodaNN [35]). When weights and activations are binarized or ternarized, MAC operations can be simplified to XNOR and pop-count operations. Fixed-bitwidth architectures replace complex MAC operations with lower-bit logic, while variable bitwidth architectures, like Stripes and UNPU, use bit-serial computations with fixed or variable bitwidths for different components. BitFusion dynamically fuses bit-level processing elements to match the bitwidth of different network layers, while BitBlade further improves on BitFusion by using bitwise summation.

D. Most Significant Digit First (MSDF) Arithmetic

Online arithmetic, also known as Most Significant Digit First (MSDF) arithmetic, is a computational approach where the processing of digits starts from the most significant position. This method is introduced by Ercegovac in [36]. In fact, the results of addition and multiplication are produced from the right to the left in conventional arithmetic [37]. However, we can generate the result from left to right i.e., the most-significant digit first via Online arithmetic. In this manner of computation, after receiving a few most significant digits of the operands (i.e., Online delay δ) the result digits are produced, as shown in Fig. 5 [38]. Therefore, the execution of the operations that are dependent on a producing-consuming arrangement can be overlapped as illustrated in Fig. 6. It provides digit-level massive parallelism in a long sequence of arithmetic operations [36].

The aforementioned capability stems from the utilization of a redundant num-

ber representation, which enables multiple representations of a given value. The value of a radix-r redundant signed-digit number X is

$$X = \sum_{i=1}^{n} x_i r^{-i} \tag{D..1}$$

where the digit set of $D = \{-b, ..., -1, 0, 1, ..., a\}$ and $a+b+1 > r$. In a redundant number representation, redundancy factor ρ is defined as $\rho = \frac{a}{r-1}$ [36]. In this paper, we use binary symmetric digit set, where $a = b$, $r = 2$, and $\rho = 1$. This number system, called Binary Signed-Digit (BSD), is represented by two same weight bits, namely a Posibit and a Negabit, to define the value of each position. It is noteworthy to mention that we use Inverted Encoding for Negabit (IEN) where a Posibit (Negabit) is represented by $\circ(\bullet)$, $x(x^-)$, $0(1^-)$, and $1(0^-)$ as a dot notation, a variable, a constant zero, and a constant one (minus one), respectively. By taking advantage of IEN, standard components such as Full Adder (FA) and Half Adder (HA) can be used without any consideration about the polarity of the input/output bits that are Negabit or Posibit, as shown in Fig. 7 [37].

1. Serial-Serial Multiplication

Ercegovac *et al.* [36] devised an Online multiplication algorithm using radix-r for n-digit operands x, y, and product p within the range of $(-1, 1)$. The operands are represented by n signed digits selected from the set $-a, ..., a$, where $a \leq r-1$.

Consider the operands and the result at cycle j to be

$$x[j] = \sum_{i=1}^{j+\delta} x_i r^{-i}, y[j] = \sum_{i=1}^{j+\delta} y_i r^{-i}, p[j] = \sum_{i=1}^{j} p_i r^{-i} \tag{D..2}$$

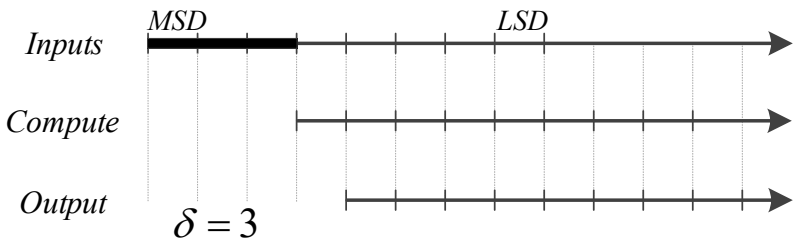


Figure 5: Timing diagram of Online operations [38].

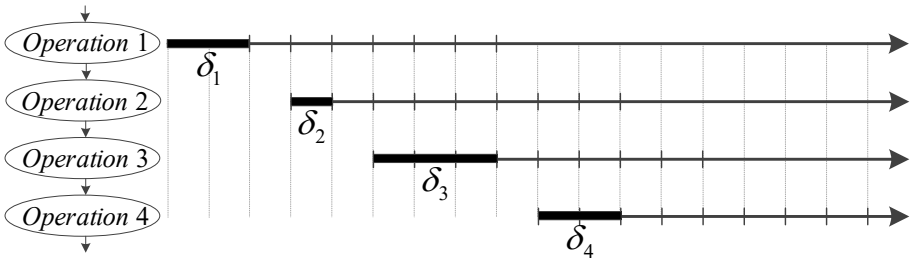


Figure 6: Timing diagram of a chain of Online operations adapted from [36].

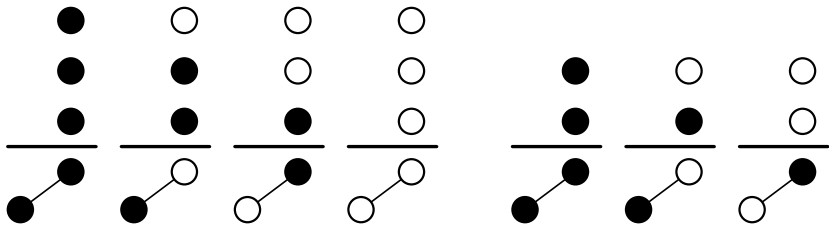


Figure 7: Functionality of FA and HA on different Posibit-Negabit mixes.

In order to form the recurrence, the residual value $w[j]$ is defined as

$$w[j] = r^j \left(x[j]y[j] - p[j]\right) \tag{D..3}$$

and $w[j+1]$ is

$$w[j+1] = rw[j] + \left(x[j]y_{j+1+\delta} + y[j+1]x_{j+1+\delta}\right) r^{-\delta} - p_{j+1} \tag{D..4}$$

This is decomposed into

$$\begin{aligned} v[j] &= rw[j] + \left(x[j]y_{j+1+\delta} + y[j+1]x_{j+1+\delta}\right) r^{-\delta} \\ w[j+1] &= v[j] - p_{j+1} \end{aligned} \tag{D..5}$$

In order to employ the carry-save representation of $w[j]$ and $v[j]$ using $\widehat{L}k$ and $\widehat{U}k$ as the interval for grid selection, the selection constants m_k are derived

$$\widehat{L}_k \leq m_k \leq \widehat{U}_{k-1} \tag{D..6}$$

where

$$\begin{aligned}\widehat{U}_k &= \left\lfloor \rho(1-2r^{-\delta})+k-2^{-t} \right\rfloor_t \\ \widehat{L}_k &= \left\lceil -\rho(1-2r^{-\delta})+k \right\rceil_t \end{aligned} \tag{D..7}$$

For a radix-2 Online multiplier with an Online delay of $\delta = 3$, $t = 2$ fractional bits, and a value of $\rho = \frac{1}{2-1} = 1$ being taken into account.

$$\begin{aligned}\widehat{U}_k &= \left\lfloor 1-2\times 2^{-3}+k-2^{-2} \right\rfloor_2 = k+2^{-1} \\ \widehat{L}_k &= \left\lceil -1+2\times 2^{-3}+k \right\rceil_2 = k-3\times 2^{-2} \end{aligned} \tag{D..8}$$

Consequently, the selection constants can be determined as follows:

$$m_0 = -2^{-1}, m_1 = 2^{-1} \tag{D..9}$$

The range of $\widehat{v}[j]$ is

$$-2 \leq \widehat{v}[j] \leq \frac{7}{4} \tag{D..10}$$

The selection function can be expressed using the following equation, as documented in [36].

$$p_{j+1} = \begin{cases} 1 & 0.5 & \leq \widehat{v}[j] \leq & 1.75 \\ 0 & -0.5 & \leq \widehat{v}[j] \leq & 0.25 \\ -1 & -2 & \leq \widehat{v}[j] \leq & -0.75 \end{cases} \tag{D..11}$$

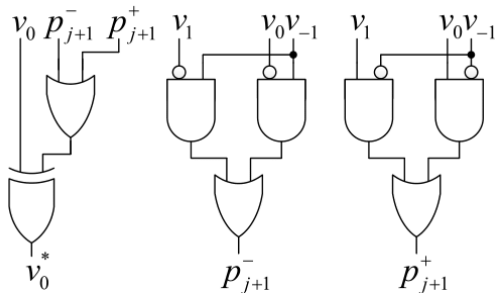


Figure 8: Required logic for complementing v_0 and generating the product.

Fig. 9 depicts the schematics of the conventional serial-serial multiplier, in which *LX* and *LY* registers are the output of the predecessor Online units. Since $w[j+1]$ has $(n+2)$ bit length, a sign extension for both selected operands are

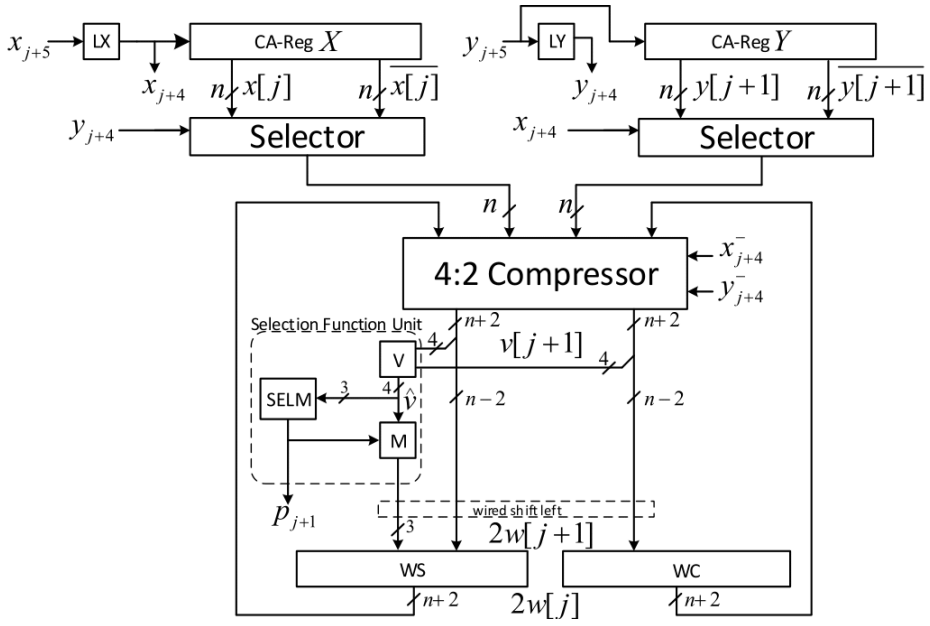


Figure 9: Conventional serial-serial Online multiplier [36].

required (i.e., for accumulation via 4:2 Compressor) [39]. The input carries of the 4:2 Compressor are associated with the signs of x_{j+4} and y_{j+4}. The module V produces the estimate of $v[j]$. Moreover, the product result (i.e., p^-_{j+1} and p^+_{j+1}) is also generated based on the estimate of $v[j]$, via a simple logic as illustrated in Fig. 8. The critical path of conventional serial-serial multiplier (See Fig. 9) consists of a Selector, a 4:2 Compressor, a 4-bit Carry Propagate Adder (i.e., V module), and 5 levels of two-input logic (one AND gate, two OR gates, and one XOR gate), which is shown in Fig. 8, for generating v^*_0. Fig. 10 depicts the order of traversal of partial product terms in the serial-serial Online multiplier. Due to the difference in the number of generated at each cycle in this design, several digit slices are not utilized. The timing diagram of the utilization of digits in this design is depicted in Fig. 11. The rate of underutilization in this design is about 50%. In the proposed computation element, we attempt to avoid this underutilization.

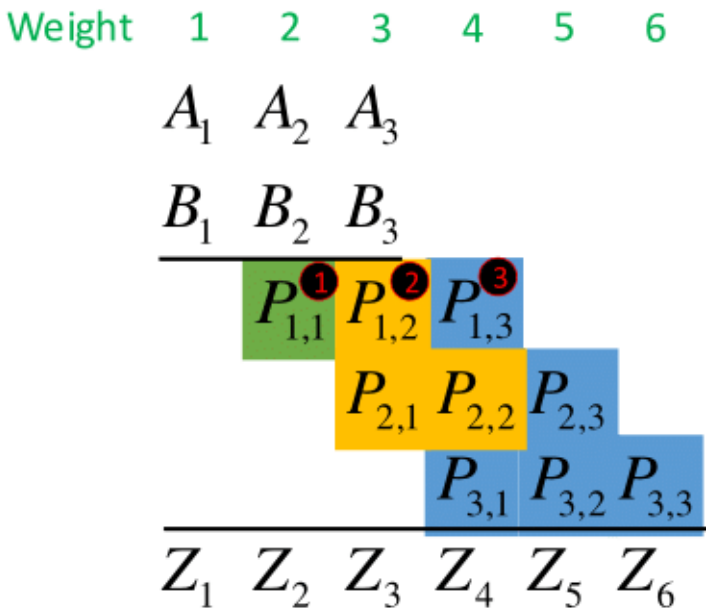


Figure 10: Traversal order of partial product terms in the serial-serial Online multiplier for $n = 3$.

2. Serial-Parallel Multiplication

An algorithm for radix-r serial-parallel Online multiplication for multiplying an n-digit radix-r signed digit operand x with a radix complement n-digit operand Y. This algorithm generates the $2n$-digit product p within the range of $(-1,1)$ is presented. Each digit of x and p is represented using signed digits selected from the set $-a,...,a$, where $a \leq r-1$. The values of x and p at cycle j are illustrated in Equation (D..2). The residual value is provided by Ercegovac in [36].

$$w[j] = r^j \left(x[j]Y - p[j]\right) \tag{D..12}$$

Therefore, the recurrence is

$$w[j+1] = rw[j] + \left(x_{j+1+\delta}Y\right) r^{-\delta} - p_{j+1} \tag{D..13}$$

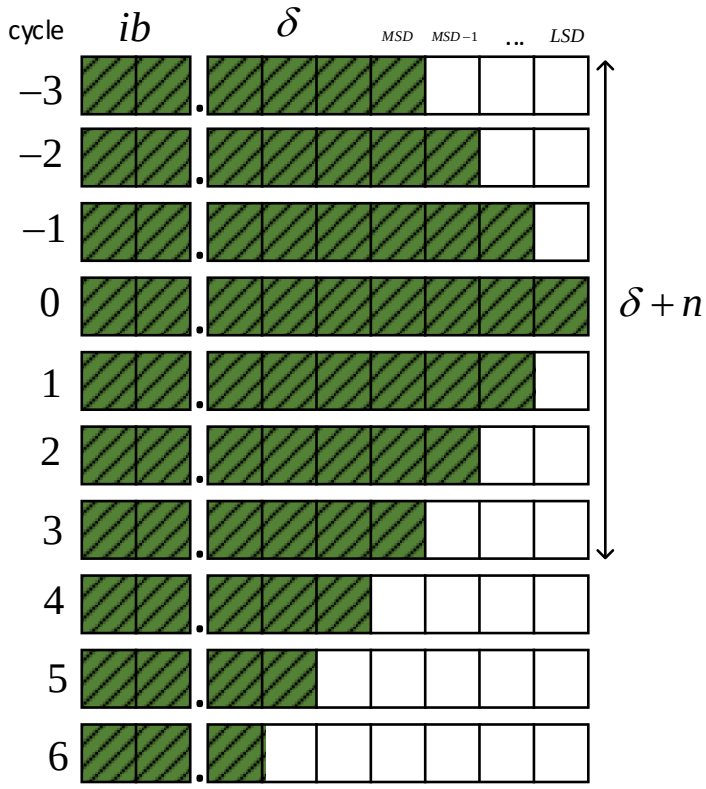


Figure 11: Digit slices underutilization in the serial-serial multiplier for $n = 3$.

This is decomposed into

$$\begin{aligned} v[j] &= rw[j] + \left(x_{j+1+\delta} Y\right) r^{-\delta} \\ w[j+1] &= v[j] - p_{j+1} \end{aligned} \tag{D..14}$$

Consider a radix-2 Online multiplier featuring an Online delay of $\delta = 2$, $t = 2$ fractional bits, and $\rho = 1$. Derived from Eq. (D..7), the value of $\widehat{L}_k$ and $\widehat{U}_k$ is

$$\begin{aligned} \widehat{U}_k &= \left\lfloor 1 - 2^{-2} + k - 2^{-2} \right\rfloor_2 = k + 2^{-1} \\ \widehat{L}_k &= \left\lceil -1 + 2^{-2} + k \right\rceil_2 = k - 3 \times 2^{-2} \end{aligned} \tag{D..15}$$

the value of $\widehat{L}_k$ and $\widehat{U}_k$ are the same as the values computed for its serial-serial counterpart. Therefore, the selection function is the same as Eq. (D..11). It should be noted that the value of δ is 2 for the serial-parallel multiplier, while

this value is 3 for the serial-serial Online multiplier. The architecture of this multiplier is illustrated in Fig. 13. In this architecture, the conversion/appending unit is not used; instead, a simple register for keeping the value of Y is required. Moreover, it has only one selector unit, and in the place of a 4:2 Compressor, a 3:2 adder is utilized. The other components remain as same as the serial-serial multiplier. Fig. 12 depicts the order of traversal of partial product terms in the serial-parallel Online multiplier.

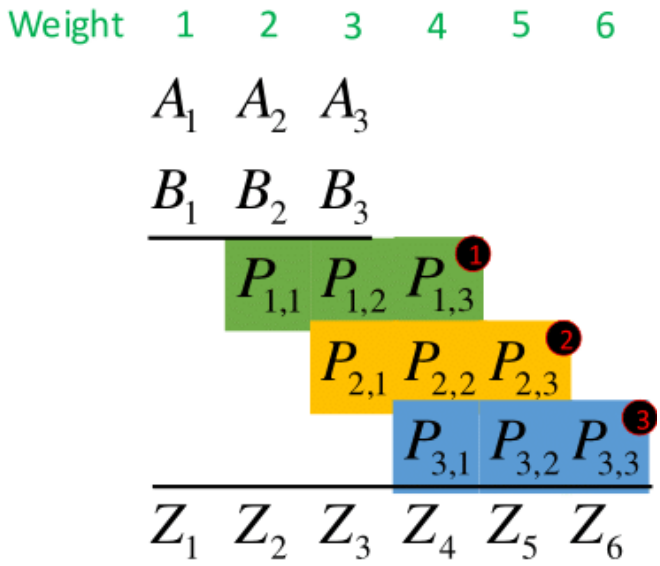


Figure 12: Traversal order of partial product terms in the serial-parallel Online multiplier for $n = 3$.

3. Inner Product

The inner product between two vectors, also known as the dot product, can be defined as:

$$p = \sum_{k=1}^{K} A_k B_k \qquad \text{(D..16)}$$

In the conventional implementation of the inner product operation in Online arithmetic, the values of A_k and B_k are multiplied using k discrete multipliers. The results of these multiplications are then added together using an Online adder tree. The multiplication operation has an Online delay of 3, while the addition operation has an Online delay of 2. The Online delay of the inner product unit

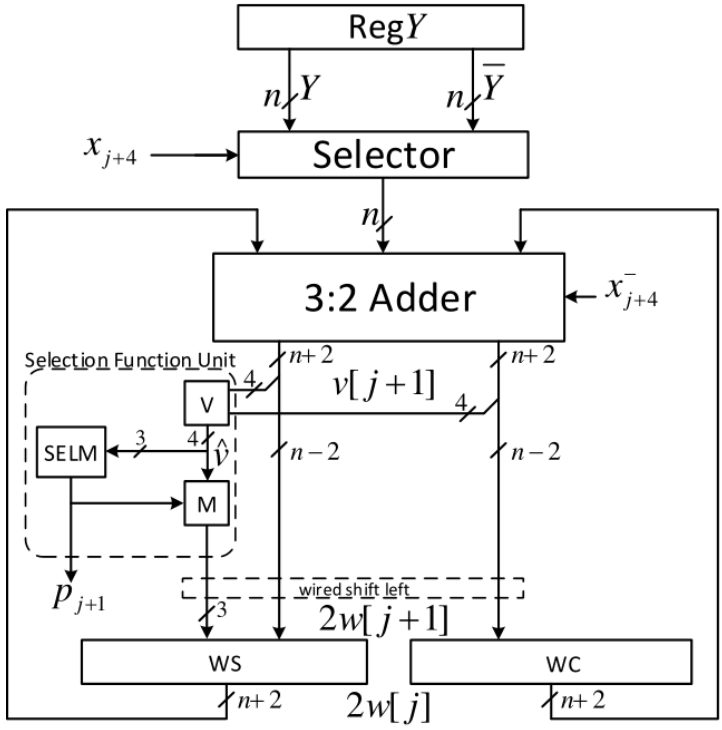
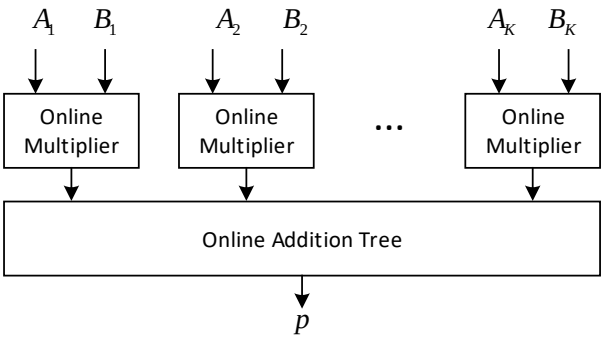


Figure 13: Conventional serial-parallel Online multiplier.

is determined by the delay of the Online multiplier plus the total delay of the Online adders in the reduction tree. Fig. 14 demonstrates the block diagram of the Online inner product unit.

4. Application of Online Arithmetic in CNN Accelerators

The reduced hardware complexity achieved through the serial nature of computation can be advantageous, and overlapping dependent operations can help offset the increased computation cycles [36, 40]. Online arithmetic is particularly suitable for variable precision computations. These characteristics make it an excellent choice for computation-intensive applications like machine learning algorithms [41, 42]. For example, online arithmetic can enable digit-level pipelining for CNN inference, as illustrated in Fig. 15. In a CNN utilizing online arithmetic, MaxPool and ReLU functions can be executed incrementally as soon as sufficient information becomes available, enabling early termination of computations for subsequent digits. This can significantly reduce the required



$$p = \sum_{k=1}^{K} A_k B_k$$

$$\delta_{Multiplication} = 3$$

$$\delta_{Addition} = 2$$

$$\delta_{Total} = \delta_{Multiplication} + \lceil \log_2^k \rceil \times \delta_{Addition}$$

Figure 14: Online inner product unit.

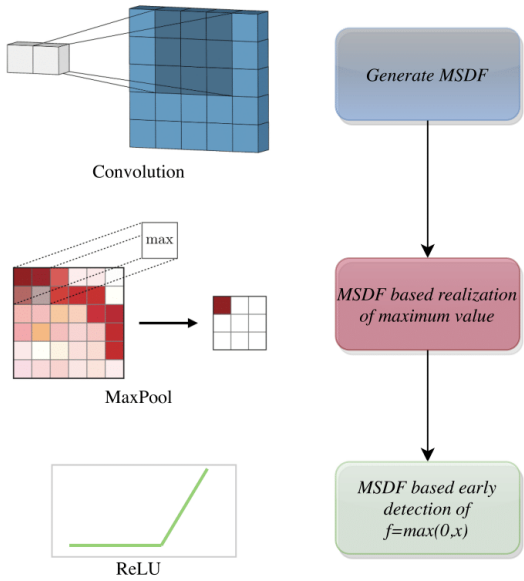


Figure 15: Online arithmetic for digit-level pipelining in CNN [8].

computation, resulting in improved processing efficiency [8].

E. Fused-Layer CNN

Fused-layer dataflow is a technique that combines the computations of multiple layers in a CNN into a single operation, which is then executed using highly optimized hardware [43]. This approach allows for significant performance gains,

as it reduces the amount of data movement required between different layers in the network, and enables highly optimized computations to be performed using specialized hardware such as GPUs and Tensor Processing Units (TPUs).

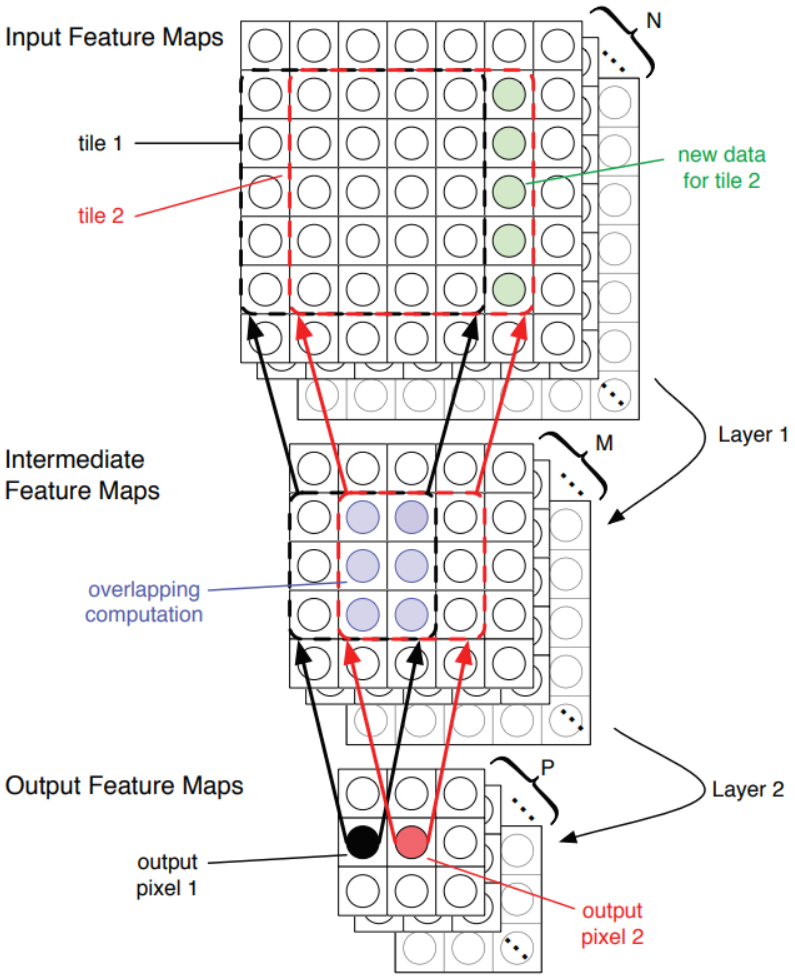


Figure 16: Example of fusing two convolutional layers [43].

The traditional approach to designing CNN accelerators focuses on processing each layer to completion, which requires off-chip memory to store intermediate data between layers. The researchers introduce a novel aspect in the design realm of CNN accelerators, concentrating on the dataflow between convolutional layers. Through altering the sequence in which input data is brought onto the chip, they showcase the capability to merge the processing of multiple CNN layers. This fusion enables the caching of intermediate data between the evaluation

of adjacent layers, leading to a significant reduction in off-chip memory bandwidth demands and minimizing data movement.

Chapter III.
The Proposed Early Termination Scheme

To avoid ineffective computations in Online arithmetic, the concept of early termination is introduced [44]. Since the digits are generated from the most significant towards the least significant, as soon as a certain condition is detected the computations for the rest of the digits can be skipped. For example, in the ReLU as soon as the negative output is detected the computations for the rest least-significant digits can be skipped.

A. Rectified Linear Unit Activation Function

The ReLU activation function can be expressed as follows:

$$ReLU(x) = max(0,x) = \begin{cases} x & if x > 0 \\ 0 & if x \leq 0 \end{cases} \quad \text{(A..1)}$$

Early Termination in ReLU occurs when the negative output is detected. In other words, when the first non-zero digit is $\bar{1}$ the rest of the computations can be skipped.

To clarify this concept two examples are provided.

Example#1 (negative output in ReLU):

$x = .00\bar{1}11111$

In this example, after visiting the third digit, $\bar{1}$, we can say that the output is definitely negative, therefore the 5 remained digits can be skipped.

Example#2 (positive output in ReLU):

$x = .01000000$

In this example, after visiting the second digit, 1, we can say that the output is definitely positive, therefore the rest of the digits should be computed.

Fig. 17 demonstrates the fuctionality of the Finite-State Machine (FSM) required for the detection of Early Termination in ReLU. The initial state is S_0 where FSM is waiting to get the first non-zero input. In this state, if the input is 0 the output ouput would be 0. If $\bar{1}$ is detected the FSM transits to state 1 where negative oupuput is detected and the rest of the computations can be skipped. If

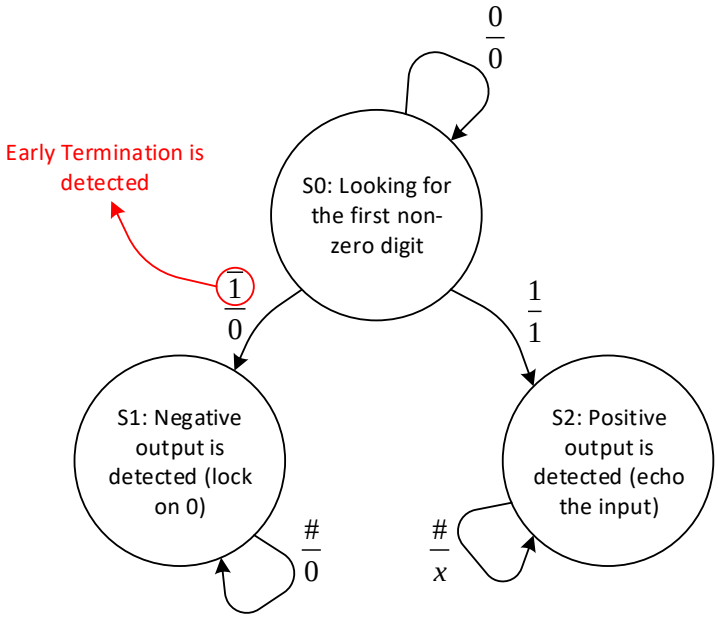


Figure 17: State-flow Diagram of Online ReLU function.

1 is detected the FSM transits to state 2 where positive oupuput is detected and the rest of the computations should be performed.

B. MaxPool Function

In CNN, the MaxPool function is described as follows:

$$y = max(X) \tag{B..1}$$

where X is a vector of m n-digit signed digit numbers.

Algorithm III..1 describes the early termination algorithm in the Online MaxPool function. In this algorithm, a boolean vector called *Effective* is assigned to indicate whether subsequent digits need to be computed. It is initialized to zero at the beginning of the algorithm. During each iteration of the algorithm, the maximum digit from all incoming digits is calculated. Each digit in each element of X is compared to the maximum digit, and if a discrepancy is found, that element of

Algorithm III..1: Online MaxPool Function Algorithm.

Data: $X[1..m][1..n]$
Result: $Effective[1..n][1..m]$

```
1 Effective[0][1..m] ← True;
2 for j ← 1 to n do
3     for i ← 1 to m do
4         maxdigit ← max(X[:][j]);
5         if Effective[j][i] then
6             if maxdigit = x[i][j] then
7                 Effective[j+1][i] ← True;
8             else
9                 Effective[j+1][i] ← False;
```

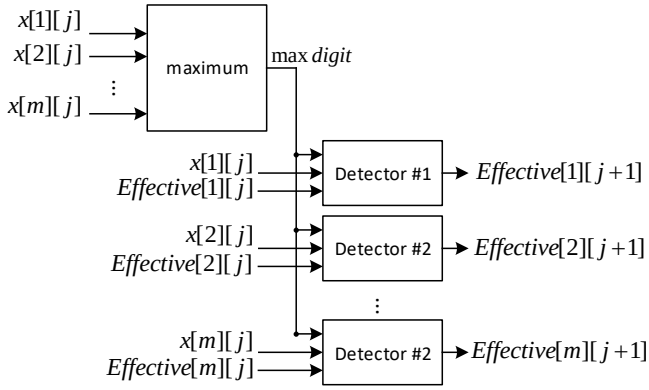


Figure 18: Block Diagram of Online MaxPool function.

X is marked as ineffective. If $Effective[i]$ becomes True in every cycle, the computations for the succeeding digits of $x[i]$ can be omitted. Fig. 18 demonstrates the hardware implementation of the proposed algorithm.

Example: Consider $m = 4$ and $n = 8$ and the vector x is as following:

$$\begin{cases} x[1] & = 0.1\bar{1}00'0000 \\ x[2] & = 0.1001'0101 \\ x[3] & = 0.1000'\bar{1}\bar{1}00 \\ x[4] & = 0.001\bar{1}'0000 \end{cases} \quad \text{(B..2)}$$

Table 1 shows the trace of signals for this example in the Online MaxPool function.

Table 1: MaxPool Early Termination Example.

j	$Effective[j+1]$	$maxdigit$
1	{True, True, True, False}	1
2	{False, True, True, False}	0
3	{False, True, True, False}	0
4	{False, True, False, False}	1
...	...	...

As shown in this Table, after receiving 4 digits of the inputs, computations for three of them are detected as ineffective. The amount of ineffective-detected digits depends on the value of pixels at run-time. In the best case, $3 \times 7 = 21$ digits can be detected as ineffective and in the worst case, the number of ineffective digits is 0. Early termination in the MaxPool function is evaluated in Chapter V.

C. Fused-layer Early Termination

By combining two layers of CNN, it becomes feasible to halt the computations of preceding layers once the output of the current layer is determined to be ineffectual. For instance, consider the sequence of layers Conv1, ReLU1, Conv2, and ReLU2. If a negative output is identified in ReLU2, all computations pertaining to the previous layers, including Conv1, ReLU1, and Conv2, can be terminated immediately.

Chapter IV.
Hardware Implementation

A. The proposed Multiplier

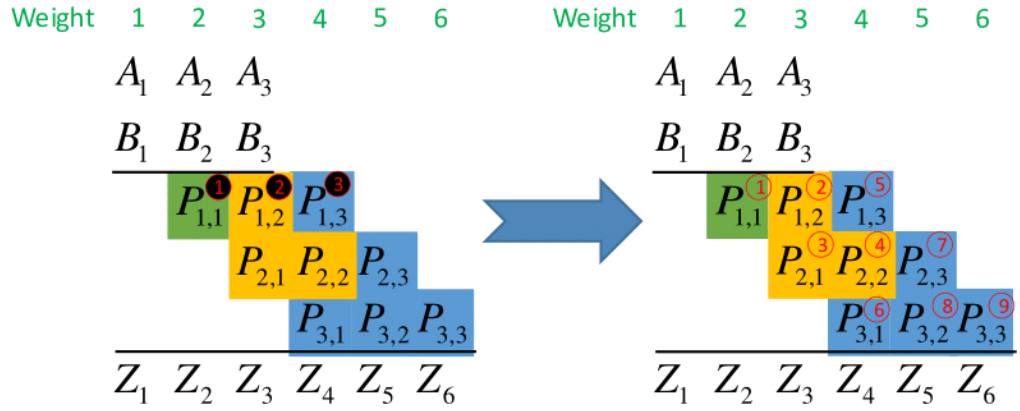
Modern hardware accelerators for CNNs offer the flexibility of using variable precision for both weights and activations, which helps in reducing unnecessary computations [32]. However, Online arithmetic, which is well-suited for CNN computations due to its variable-precision nature, suffers from a significant amount of hardware underutilization in the Online multipliers.

To address this issue, serial-parallel multipliers can be employed [45]. However, these multipliers have one input connected in parallel, which is not compatible with the nature of convolutional layers in CNNs. This limitation leads to inefficient computations. For instance, if a hardware accelerator has one of its input operands fixed at 16 bits but only a 5-bit operand is required for the computation, 11 bits are wasted on ineffective computations.

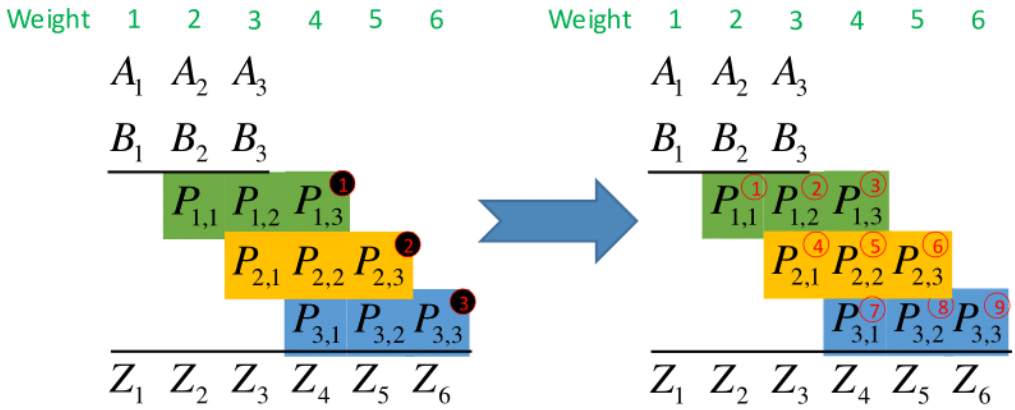
In our proposed implementation of the Online multiplication algorithm, our goal is to optimize hardware utilization. To achieve this, we make adjustments to the generation of partial product terms by ensuring a constant bit width for each term to be added to the residual in every cycle. This approach helps avoid the underutilization of hardware resources. When it comes to iterating through the partial product terms, there are two options to consider: the order of generation in a serial-serial multiplier or a serial-parallel multiplier. Figure 19(a) and 19(b) illustrate the possible traversal orders for the partial product terms in a serial-serial and serial-parallel multiplier, respectively.

In a serial-serial Online multiplication algorithm, the Online delay (δ) is 3, whereas the Online delay of a serial-parallel multiplication algorithm is 2. As shown in Figure 19, once 9 partial product terms are generated in the serial-serial multiplication algorithm and 6 partial product terms are generated in the serial-parallel multiplication algorithm, the first digit of the output is generated.

To generate the weights of the partial product terms in the proposed multiplier, a variable shifter (i.e., barrel shifter) is necessary. However, if the serial-parallel multiplication algorithm is adapted for the proposed multiplier with a constant left shift, the weights of the partial product terms can be generated ef-



(a) serial-serial partial product traversal order



(b) serial-parallel partial product traversal order

Figure 19: The traversal orders for the partial product terms in a serial-serial (a) and serial-parallel Online multiplier.

ficiently. Therefore, to design the proposed multiplier, we choose to adapt the serial-parallel multiplication algorithm.

To generate each row of partial product terms, represented by different colors in Figure 19(b), a register called Partial Product Row (PPR) is utilized. Additionally, an additional register is required to store the residual in the online algorithm. The micro-operations for each cycle are illustrated in Figure 20.

In the proposed implementation, every n cycles correspond to one iteration of the Online multiplication algorithm. During the remaining cycles, each row of the partial product array is generated by shifting the previous value stored in the

PPR register and adding a new partial product term denoted as $P_{i,j}$.

Cycle	PPG	Operation	PPR
0	1	PPR* = $P_{1,1}$	0
1	2	PPR* = $P_{1,2}$ + 2*PPR	$P_{1,1}$
2	3	Residual* = $P_{1,3}$ + 2*Residual + 2*PPR	$2P_{1,1} + P_{1,2}$
3	4	PPR* = $P_{2,1}$	0
4	5	PPR* = $P_{2,1}$ + 2*PPR	$P_{2,1}$
5	6	Residual* = $P_{2,3}$ + 2*Residual + 2*PPR – Z_1	$2P_{2,1} + P_{2,2}$
6	7	PPR* = $P_{3,1}$	0
7	8	PPR* = $P_{3,1}$ + 2*PPR	$P_{3,1}$
8	9	Residual* = $P_{3,3}$ + 2*Residual + 2*PPR – Z_2	$2P_{3,1} + P_{3,2}$
9	-	Residual* = 2*Residual – Z_3	Don't Care

Figure 20: Proposed Online multiplier trace at each cycle for $n = 3$.

The block diagram of the proposed multiplier is depicted in Fig. 21. To perform the addition of the residual register and PPR register with the new partial product term, a 6:2 compressor is employed.

Both the residual register and PPR register require a bit width of $2 \times W$ since their values are stored in carry-save format to prevent carry propagation during addition. As the residual register is updated every n cycles, its value should not be added in those cycles. Thus, a multiplexer is incorporated into the path of the residual value.

Similarly, the PPR register needs to be reset every n cycles. To achieve this, a multiplexer is used to input a zero value instead of the previous PPR value.

Additionally, the residual and PPR registers have enabled signals to ensure that the output of the 6:2 compressor is loaded only at the correct time.

B. The proposed Inner Product Unit

The inner product function is expressed as:

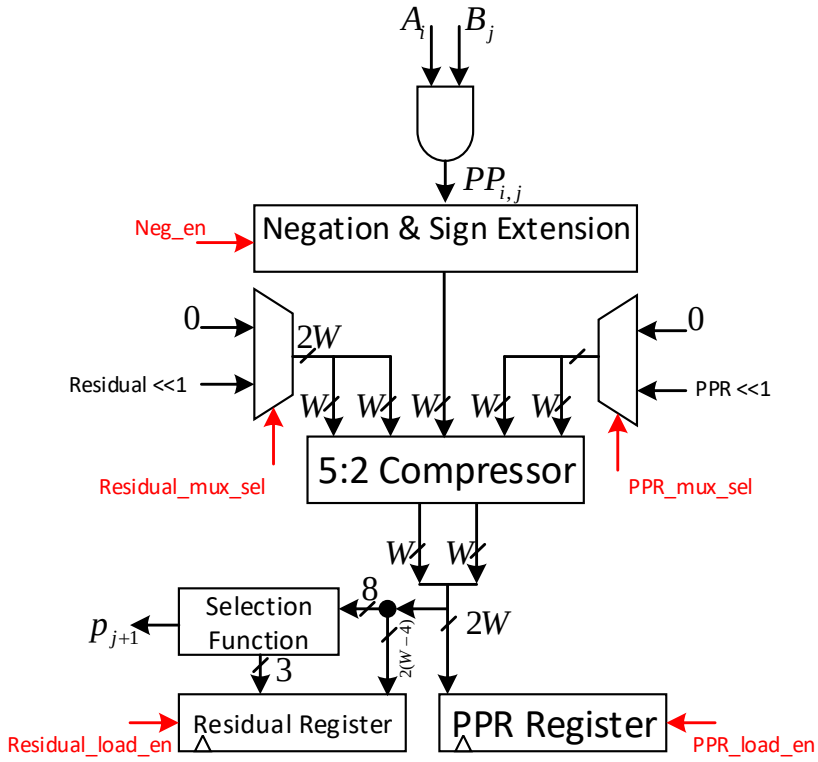


Figure 21: Proposed Online Multiplier.

$$p = \sum_{k=1}^{K} A_k B_k \tag{B..1}$$

Since performing k discrete multiplications and accumulating their results in a Online reduction tree impose a high cost for the residual and PPR registers, all of the multiplications are merged in a single inner product unit. By expanding A_k and B_k in Eq. B..1:

$$p = \sum_{k=1}^{K} \sum_{i=1}^{n} A_{k,i} 2^{-i} \sum_{j=1}^{n} B_{k,j} 2^{-j} \tag{B..2}$$

By rearranging the order of the summation operators, we obtain:

$$p = \sum_{i=1}^{n} \sum_{j=1}^{n} \sum_{k=1}^{K} A_{k,i} B_{k,j} 2^{-(j+i)} \tag{B..3}$$

In a single cycle, each partial product term of k multiplications ($A_{k,i}B_{k,j}$) is

generated. This means that k multiplication operations are combined together, as illustrated in Fig. 22. If the input data (i.e., A_k's and B_k's) is stored in the conventional format in the memory, it needs to be converted into the bit-serial format. To eliminate the need for parallel-to-serial converters, the inputs are stored in a transposed manner, as depicted in Fig. 23.

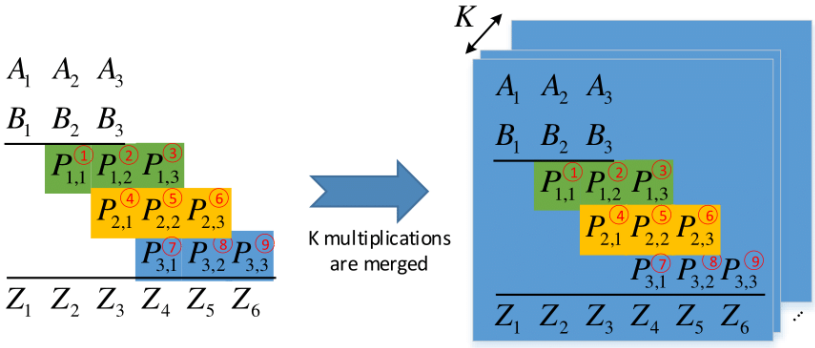


Figure 22: Proposed technique for merging k multiplications.

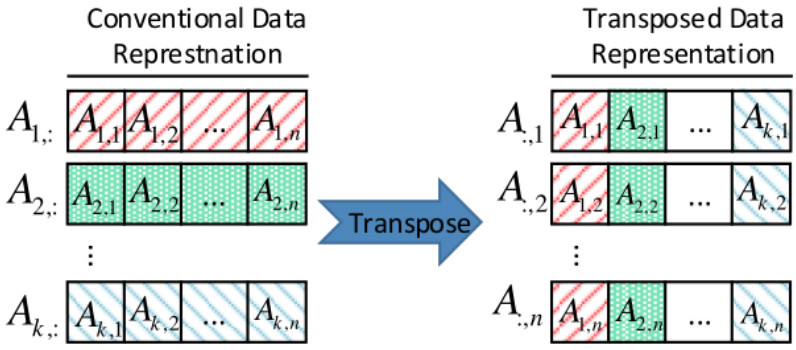


Figure 23: Transposition of activations and weights to avoid serial-parallel conversion.

The Online inner product unit, proposed alongside the Online multiplier, operates in a similar manner. However, there is a distinction in how they function. Rather than producing a single partial product term for each multiplication operation, the Online inner product unit simultaneously generates k partial product terms for k multiplications. These terms are then accumulated using the popcount circuit, as depicted in Fig. 24. The control signals and their timing schedule align with those of the proposed multiplier, as illustrated in Fig. 20.

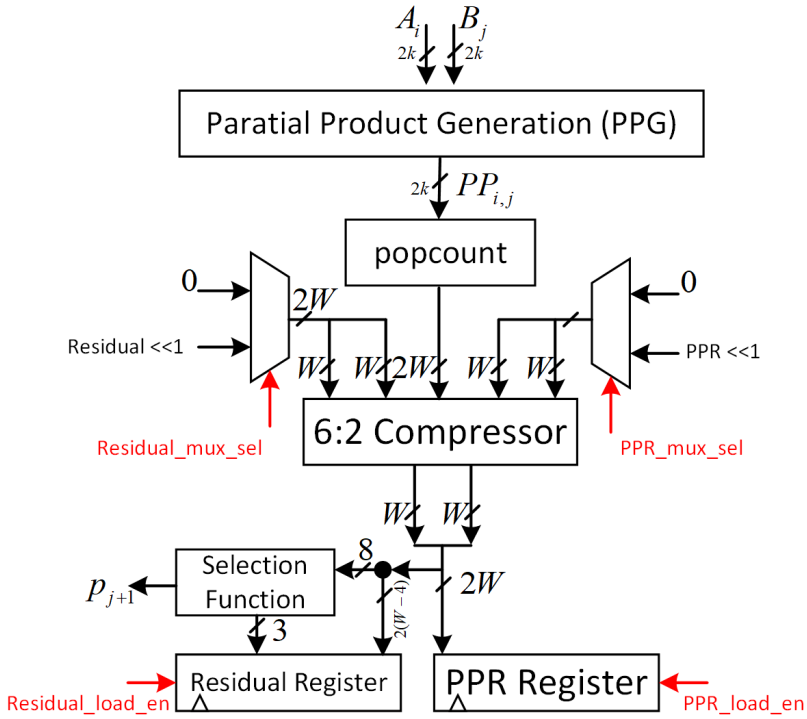


Figure 24: Proposed inner product unit.

The proposed Online inner product unit (Fig. 24) and the conventional Online inner product unit (Fig. 14) can be compared based on several key factors.

In the proposed Online inner product unit:

- The inputs are stored in the buffer and accessed multiple times.
- It utilizes smaller hardware but requires more cycles for computations.
- This design aims for nearly 100% utilization of hardware resources.
- It employs a complex control unit using micro-code.
- As a result, it achieves higher throughput within the same area.

On the other hand, the conventional Online inner product unit has different characteristics:

- The inputs are converted to 2's complement numbers, which imposes high hardware cost.

- It requires larger hardware but completes computations in fewer cycles.
- The utilization of hardware resources is approximately 50%.
- The control unit in this unit is simpler compared to the proposed design.
- It offers better latency performance.

These distinctions highlight the trade-offs between the two units. The proposed design utilizes smaller hardware efficiently and achieves higher throughput but at the cost of more cycles for computations and a complex control unit. On the other hand, the conventional design requires larger hardware but provides better latency performance with a simpler control unit.

Chapter V.

Evaluation

In this chapter, firstly, the impact of the proposed early termination scheme on the throughput of the computation units is evaluated. Then, the proposed inner product unit is compared with its conventional counterpart in terms of area, power, throughput, throughput per area, and throughput per power.

A. Experimental Setup

The architecture of the evaluated neural network (LeNet-5) is indicated in Table 2. Pytorch library in Python is used to implement LeNet-5. The network's weights are trained and quantized using Pytorch quantization library. C++ programming language is used to model the functionality of online components. Lastly, the feed-forward call of the network is implemented using online arithmetic components.

Table 2: LeNet-5 architecture.

Layer	Input Size	Output Size	Kernel
1	1x32x32	6x28x28	Convolution(kernel_size = 5x5, in_channels = 1, out_channels = 6)
1	6x28x28	6x28x28	ReLU
1	6x28x28	6x14x14	MaxPool(kernel_size = 2)
2	6x14x14	16x10x10	Convolution(kernel_size = 5x5, in_channels = 6, out_channels = 16)
2	16x10x10	16x10x10	ReLU
2	16x10x10	16x4x4	MaxPool(kernel_size = 2)
-	16x4x4	256	Flatten
3	256	120	FullyConnected
3	120	120	ReLU
4	120	84	FullyConnected
4	84	84	ReLU
5	84	10	FullyConnected
5	10	10	Softmax

To evaluate the hardware performance, both inner product units are implemented in SystemVerilog, subjected to exhaustive testing, and synthesized using the Synopsys Design Compiler with the 45*nm* FreePDK gate library. To better study the impact of the size of the problem on its performance metrics, different number of input operands (i.e., 8, 16, 32, 64, 128, 256, 512, and 1024) are evaluated. In our synthesis experimental setup, we swept the constrained clock period from 2.4 (*ns*) down to 0.5 (*ns*) by 0.1 (*ns*) intervals for better analysis of

Table 3: Early termination rate in LeNet-5 for ReLU.

Layer	Skipped Bits		Computed Bits		Total	
	#	%	#	%	#	%
Layer #0	21053	7.61	255427	92.39	276480	100.00
Layer #1	23555	28.75	58365	71.25	81920	100.00

Table 4: Early termination bit-level profile for LeNet-5 for ReLU.

Layer		Layer #1	Layer #2
Zero	#	16755	222
	%	48.48	2.17
Positive	#	12217	4338
	%	35.35	42.36
Negative(ET at bit 0)	#	184	29
	%	0.53	0.28
Negative(ET at bit 1)	#	825	630
	%	2.39	6.15
Negative(ET at bit 2)	#	1193	1909
	%	3.45	18.64
Negative(ET at bit 3)	#	1104	1655
	%	3.19	16.16
Negative(ET at bit 4)	#	906	847
	%	2.62	8.27
Negative(ET at bit 5)	#	645	348
	%	1.87	3.40
Negative(ET at bit 6)	#	426	170
	%	1.23	1.66
Negative(ET at bit 7)	#	305	92
	%	0.88	0.90
Total	#	34560	10240
	%	100.00	100.00

the effect of the input dimension on the critical path delay. The area and power reports are based on the synthesis results of the fastest designs. Other parameters (i.e., throughput, throughput per area, and throughput per power) are computed based on the new area and power values. In addition, a switching activity (α) of 0.5 is taken into account to measure power consumption.

B. Bit-level Analysis

Table 3 reports the rate of skipped bits in the computation when using the proposed technique in ReLU. The bit-level profile of early termination is demonstrated in Table 4. This information is demonstrated in Table 5 for the fused ReLU and MaxPool layers together, which is called inter-layer fusing. Table 6

Table 5: Early termination rate for inter-layer fusing.

Layer	Skipped Bits		Computed Bits		Total	
	#	%	#	%	#	%
Layer #0	62289	22.53	214191	77.47	276480	100.00
Layer #1	34135	41.67	47785	58.33	81920	100.00

depicts the rate of skipped bits and possible speedup when fusing two first convolutional layers of LeNet-5, which is called intra-layer fusing, as demonstrated in Table 2. Also, the bit-level profile of early termination is demonstrated in Table 4. This information is demonstrated in Table 5 for the inter-layer fusing.

Table 6: Early termination rate for intra-layer fusing.

Layer	Skipped Bits		Computed Bits		Total	
	#	%	#	%	#	%
Layer #0	2.34E+07	56.92	1.77E+07	43.08	4.11E+07	100.00
Layer #1	6.23E+04	22.53	2.14E+05	77.47	2.76E+05	100.00
Layer #0 + Layer #1	2.35E+07	56.69	1.79E+07	43.31	4.14E+07	100.00

Table 6 depicts the rate of skipped bits and possible speedup of intra-layer fusing as demonstrated in Table 2.

Due to the nature of the redundant number system, the proposed MaxPool function has an approximation property. Table 7 shows the rate of mismatch between the exact and approximate MaxPool function which is about 1% and can be neglected.

Table 7: Comparison between exact and approximate MaxPool function.

Layer	Matched Outputs		Different Outputs		Total	
	#	%	#	%	#	%
Layer #0	8547	98.92	93	1.08	8640	100.00
Layer #1	2532	98.91	28	1.09	2560	100.00

C. Hardware Evaluation

The proposed inner product unit and its conventional counterpart are implemented with the bit precision of 8 and the number of input operands (k in Eq. B..1 in the chapter IV.) of 8, 16, 32, 64, 128, 256, 512, and 1024. The area, power, and critical path delay of these two designs are compared and shown in Table 8 (Fig. 25), Table 9 (Fig. 26), and Table 10, respectively.

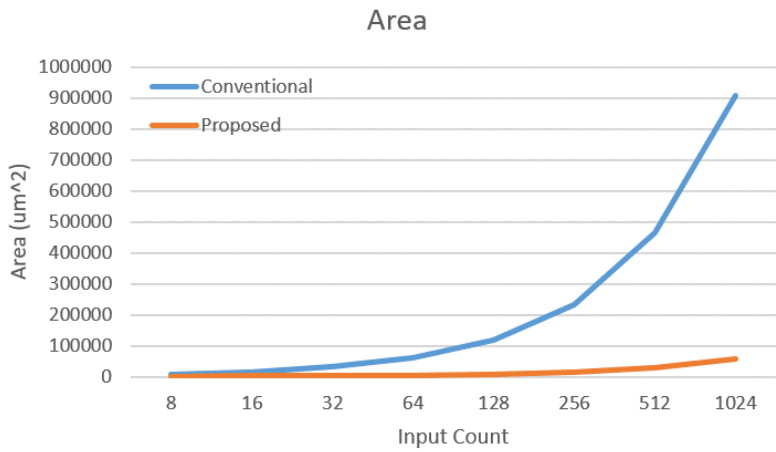


Figure 25: Comparison of the area between the proposed and conventional inner product units.

Table 8: Comparison of the area between the proposed and conventional inner product units.

Design	Area(um^2)							
	8	16	32	64	128	256	512	1024
Conventional	8229.94	16038.48	31217.16	60873.47	118703.27	233845.44	463013.96	907507.38
Proposed	1571.13	2191.13	3167.39	4873.94	8005.66	16179.64	30171.11	57325.12

The conventional design achieves a consistent delay of 0.7 *ns* as the optimal clock period. This is because the online multipliers in the conventional inner product unit have a fixed critical path delay. Even if the inner product unit size (i.e., k in Eq. B..1 in the chapter IV.) increases, the number of multipliers increases but the critical path delay remains unaffected. On the other hand, the critical path delay of the proposed design depends on the design size (i.e., k in Eq. B..1 in the chapter IV.).

Table 9: Comparison of the power consumption between the proposed and conventional inner product units.

Design	Power($mWatt$)							
	8	16	32	64	128	256	512	1024
Conventional	13.96	28.09	56.13	110.57	215.61	424.75	841.00	1648.35
Proposed	1.29	1.39	1.77	2.34	4.23	5.66	9.43	16.03

The size of the conventional design exhibits a linear correlation with the number of inputs due to the increase in the number of multipliers, which account for the majority of the design's area. As the number of input operands increases, the number of multipliers grows linearly, thereby impacting the overall area. On the other hand, in the proposed inner product unit, the utilization of shared registers and 6:2 compressors for all multipliers results in a lower rate of area increase

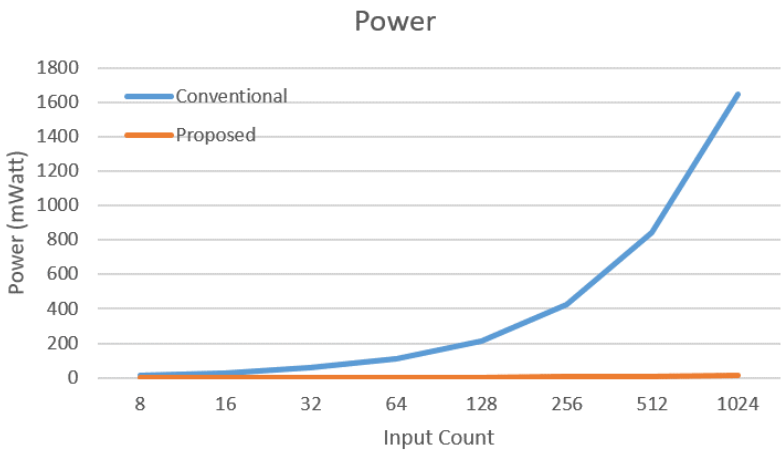


Figure 26: Comparison of the area between the proposed and conventional inner product units.

compared to the conventional design.

The power of each designs are proportional to the area of each of them. Since, the conventional design has higher clock frequency for input count of 32, 64, 128, 256, 512, and 1024, the dynamic power of the proposed design is lower than the conventional deign in the same area. This difference can be seen in the Fig 27 and 28, where the ratio of throughput per power of the proposed design to throughput per power of the conventional design is higher than the ratio of the throughput per area of the proposed design to the throughput per area of the conventional design.

Table 10: Comparison of the timing between the proposed and conventional inner product units.

Design	Clock Period(ns)							
	8	16	32	64	128	256	512	1024
Conventional	0.7	0.7	0.7	0.7	0.7	0.7	0.7	0.7
Proposed	0.6	0.7	0.8	0.9	1.0	1.4	1.8	2.2

The number of computation cycles is computed using Eq. C..1 and Eq. C..2 and reported in Table 11.

$$\delta_{Conventional} = n + \delta_{Multiplication} + \lceil log^{k} 2 \rceil \times \delta_{Addition} \tag{C..1}$$

$$\delta_{Proposed} = n^{2} + \delta_{Multiplication} \tag{C..2}$$

The computation time of each design is computed using Eq. C..3 and reported

Table 11: Comparison of the computation cycles between the proposed and conventional inner product units.

Design	Cycles (#)							
	8	16	32	64	128	256	512	1024
Conventional	17	19	21	23	25	27	29	31
Proposed	66	66	66	66	66	66	66	66

in Table 12.

$$Time = Cycles \times T_{clock} \tag{C..3}$$

The throughput of inner product units is computed using Eq. C..4 and shown in Table 13.

$$Throughput(GOp/s) = \frac{1(Op)}{Time(ns)} \tag{C..4}$$

Table 12: Comparison of the computation time between the proposed and conventional inner product units.

Design	Time (ns)							
	8	16	32	64	128	256	512	1024
Conventional	1.19E+01	1.33E+01	1.47E+01	1.61E+01	1.75E+01	1.89E+01	2.03E+01	2.17E+01
Proposed	3.96E+01	4.62E+01	5.28E+01	5.94E+01	6.60E+01	9.24E+01	1.19E+02	1.45E+02

Table 13: Comparison of the throughput between the proposed and conventional inner product units.

Design	Throughput (GOp/s)							
	8	16	32	64	128	256	512	1024
Conventional	8.40E-02	7.52E-02	6.80E-02	6.21E-02	5.71E-02	5.29E-02	4.93E-02	4.61E-02
Proposed	2.53E-02	2.16E-02	1.89E-02	1.68E-02	1.52E-02	1.08E-02	8.42E-03	6.89E-03

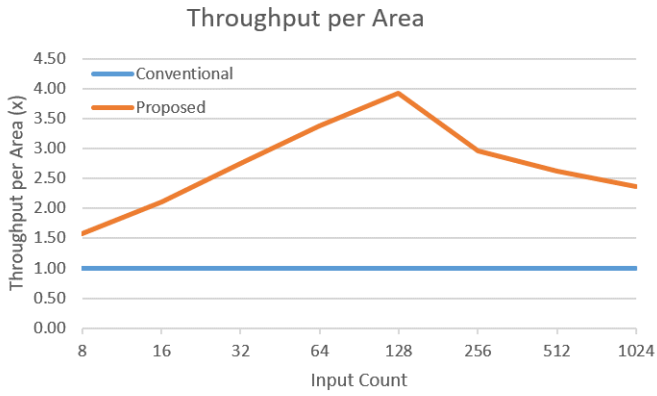


Figure 27: Comparison of the throughput per area between the proposed and conventional inner product units.

Table 14: Comparison of the throughput per area between the proposed and conventional inner product units.

Design	Throughput per Area ($GOp/s.um^2$)							
	8	16	32	64	128	256	512	1024
Conventional	1.02E-05	4.69E-06	2.18E-06	1.02E-06	4.81E-07	2.26E-07	1.06E-07	5.08E-08
Proposed	1.61E-05	9.88E-06	5.98E-06	3.45E-06	1.89E-06	6.69E-07	2.79E-07	1.20E-07

In order to have a fair comparison between the proposed and conventional designs, the throughput of them are normalized based on the area and power and reported in Table 14 and Table 15, respectively. The ratio of the throughput per area of the proposed design to the throughput per area of the conventional design is shown in Fig. 27. As the input dimension (i.e., k) increases, the proposed design experiences an increase in the critical path delay. However, the throughput per area of the design does not grow proportionally across all dimensions.

The relationship between the throughput per power and the throughput per area in the proposed design follows a similar pattern. The ratio between the throughput per power of the proposed design and that of the conventional design is depicted in Fig. 28. As the input dimension (i.e., k) expands, the proposed design encounters an upsurge in the critical path delay.

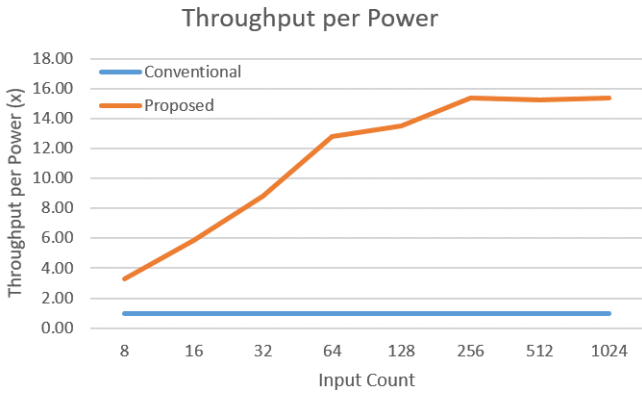


Figure 28: Comparison of the throughput per power between the proposed and conventional inner product units.

Table 15: Comparison of the throughput per power between the proposed and conventional inner product units.

Design	Throughput per Power ($GOp/s.mWatt$)							
	8	16	32	64	128	256	512	1024
Conventional	6.02E-03	2.68E-03	1.21E-03	5.62E-04	2.65E-04	1.25E-04	5.86E-05	2.80E-05
Proposed	1.96E-02	1.56E-02	1.07E-02	7.21E-03	3.58E-03	1.91E-03	8.93E-04	4.30E-04

Chapter VI.

Conclusion

This thesis proposed a hardware accelerator for Convolutional Neural Networks (CNNs) using Most-Significant-Digit First (MSDF) arithmetic and fused-layer dataflow techniques. The proposed accelerator enables digit-level pipelining across successive layers while terminating ineffective computations early, resulting in significant improvements in both computational efficiency and memory requirements. The evaluation results demonstrate that up to 56.3% of computations can be skipped when the first two layers of LeNet-5 are fused together.

The conclusions drawn from this thesis have noteworthy implications for the fields of deep learning hardware accelerator design. The proposed accelerator can be used in a wide range of applications, from image and video processing to natural language processing and speech recognition. It can also potentially reduce the cost and energy consumption of deep learning systems, which can make them more accessible and affordable.

Future work could focus on optimizing the hardware implementation of the proposed accelerator to further improve its performance and efficiency. Moreover, the proposed techniques could be extended to other types of neural networks and arithmetic operations, opening up new possibilities for hardware acceleration in the field of deep learning.

Bibliography

[1] W. Samek, G. Montavon, S. Lapuschkin, C. J. Anders, and K.-R. Müller, "Explaining deep neural networks and beyond: A review of methods and applications," *Proceedings of the IEEE*, vol. 109, no. 3, pp. 247–278, 2021.

[2] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Communications of the ACM*, vol. 60, no. 6, pp. 84–90, 2017.

[3] F. Conti and L. Benini, "A ultra-low-energy convolution engine for fast brain-inspired vision in multicore clusters," in *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2015, pp. 683–688.

[4] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *The bulletin of mathematical biophysics*, vol. 5, pp. 115–133, 1943.

[5] L. Alzubaidi, J. Zhang, A. J. Humaidi, A. Al-Dujaili, Y. Duan, O. Al-Shamma, J. Santamaría, M. A. Fadhel, M. Al-Amidie, and L. Farhan, "Review of deep learning: Concepts, cnn architectures, challenges, applications, future directions," *Journal of big Data*, vol. 8, pp. 1–74, 2021.

[6] D. Blalock, J. J. Gonzalez Ortiz, J. Frankle, and J. Guttag, "What is the state of neural network pruning?" *Proceedings of machine learning and systems*, vol. 2, pp. 129–146, 2020.

[7] M. Alwani, H. Chen, M. Ferdman, and P. Milder, "Fused-layer cnn accelerators," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–12.

[8] A. S. Hassan, T. Arifeen, and J.-A. Lee, "Data footprint reduction in dnn inference by sensitivity-controlled approximations with online arithmetic," in *2020 23rd Euromicro Conference on Digital System Design (DSD)*. IEEE, 2020, pp. 534–541.

[9] S. Herculano-Houzel, "The human brain in numbers: a linearly scaled-up primate brain," *Frontiers in human neuroscience*, p. 31, 2009.

[10] R. K. Srivastava, K. Greff, and J. Schmidhuber, "Training very deep networks," *Advances in neural information processing systems*, vol. 28, 2015.

[11] V. G. Maltarollo, K. M. Honório, and A. B. F. da Silva, "Applications of artificial neural networks in chemical problems," *Artificial neural networks-architectures and applications*, pp. 203–223, 2013.

[12] A. Krenker, J. Bešter, and A. Kos, "Introduction to the artificial neural networks," *Artificial Neural Networks: Methodological Advances and Biomedical Applications. InTech*, pp. 1–18, 2011.

[13] N. McClure, *TensorFlow machine learning cookbook*. PACKT publishing Ltd, 2017.

[14] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," *Advances in neural information processing systems*, vol. 28, 2015.

[15] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv preprint arXiv:1510.00149*, 2015.

[16] M. Zhu and S. Gupta, "To prune, or not to prune: exploring the efficacy of pruning for model compression," *arXiv preprint arXiv:1710.01878*, 2017.

[17] T. Wang, K. Wang, H. Cai, J. Lin, Z. Liu, H. Wang, Y. Lin, and S. Han, "Apq: Joint search for network architecture, pruning and quantization policy," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020, pp. 2078–2087.

[18] S. Jung, C. Son, S. Lee, J. Son, J.-J. Han, Y. Kwak, S. J. Hwang, and C. Choi, "Learning to quantize deep networks by optimizing quantization intervals with task loss," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019, pp. 4350–4359.

[19] Z. Liu, M. Sun, T. Zhou, G. Huang, and T. Darrell, "Rethinking the value of network pruning," *arXiv preprint arXiv:1810.05270*, 2018.

[20] H. Wu, P. Judd, X. Zhang, M. Isaev, and P. Micikevicius, "Integer quantization for deep learning inference: Principles and empirical evaluation," *arXiv preprint arXiv:2004.09602*, 2020.

[21] H. Cai, C. Gan, T. Wang, Z. Zhang, and S. Han, "Once-for-all: Train one network and specialize it for efficient deployment," *arXiv preprint arXiv:1908.09791*, 2019.

[22] X. Xia, X. Xiao, X. Wang, and M. Zheng, "Progressive automatic design of search space for one-shot neural architecture search," in *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision*, 2022, pp. 2455–2464.

[23] M. Tan, B. Chen, R. Pang, V. Vasudevan, M. Sandler, A. Howard, and Q. V. Le, "Mnasnet: Platform-aware neural architecture search for mobile," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2019, pp. 2820–2828.

[24] M. S. Abdelfattah, Ł. Dudziak, T. Chau, R. Lee, H. Kim, and N. D. Lane, "Best of both worlds: Automl codesign of a cnn and its hardware accelerator," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2020, pp. 1–6.

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

[26] X. Zhou, Z. Du, Q. Guo, S. Liu, C. Liu, C. Wang, X. Zhou, L. Li, T. Chen, and Y. Chen, "Cambricon-s: Addressing irregularity in sparse neural networks through a cooperative software/hardware approach," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 15–28.

[27] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "Scnn: An accelerator for

compressed-sparse convolutional neural networks," *ACM SIGARCH computer architecture news*, vol. 45, no. 2, pp. 27–40, 2017.

[28] Y.-H. Chen, T.-J. Yang, J. Emer, and V. Sze, "Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 9, no. 2, pp. 292–308, 2019.

[29] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE journal of solid-state circuits*, vol. 52, no. 1, pp. 127–138, 2016.

[30] J.-F. Zhang, C.-E. Lee, C. Liu, Y. S. Shao, S. W. Keckler, and Z. Zhang, "Snap: An efficient sparse neural acceleration processor for unstructured sparse deep neural network inference," *IEEE Journal of Solid-State Circuits*, vol. 56, no. 2, pp. 636–647, 2020.

[31] P. Judd, J. Albericio, T. Hetherington, T. M. Aamodt, and A. Moshovos, "Stripes: Bit-serial deep neural network computing," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–12.

[32] H. Sharma, J. Park, N. Suda, L. Lai, B. Chau, V. Chandra, and H. Esmaeilzadeh, "Bit fusion: Bit-level dynamically composable architecture for accelerating deep neural network," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018, pp. 764–775.

[33] J. Lee, C. Kim, S. Kang, D. Shin, S. Kim, and H.-J. Yoo, "Unpu: An energy-efficient deep neural network accelerator with fully variable weight bit precision," *IEEE Journal of Solid-State Circuits*, vol. 54, no. 1, pp. 173–185, 2018.

[34] S. Ryu, H. Kim, W. Yi, and J.-J. Kim, "Bitblade: Area and energy-efficient precision-scalable neural network accelerator with bitwise summation," in

Proceedings of the 56th Annual Design Automation Conference 2019, 2019, pp. 1–6.

[35] R. Andri, L. Cavigelli, D. Rossi, and L. Benini, "Yodann: An architecture for ultralow power binary-weight cnn acceleration," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 1, pp. 48–60, 2017.

[36] M. D. Ercegovac and T. Lang, *Digital arithmetic*. Elsevier, 2004.

[37] B. Parhami, *Computer arithmetic*. Oxford university press, 2010, vol. 20, no. 00.

[38] M. D. Ercegovac, "On left-to-right arithmetic," in *2017 51st Asilomar Conference on Signals, Systems, and Computers*. IEEE, 2017, pp. 750–754.

[39] M. Ercegovac and T. Lang, "On-line scheme for normalizing a 3-d vector," in *Conference Record of the Thirty-Third Asilomar Conference on Signals, Systems, and Computers (Cat. No. CH37020)*, vol. 2. IEEE, 1999, pp. 1460–1464.

[40] A. M. Abdelhadi and L. Shannon, "Revisiting Deep Learning Parallelism: Fine-Grained Inference Engine Utilizing Online Arithmetic," in *2019 International Conference on Field-Programmable Technology (ICFPT)*. IEEE, 2019, pp. 383–386.

[41] S. Gorgin, M. H. Gholamrezaei, D. Javaheri, and J.-A. Lee, "An efficient fpga implementation of k-nearest neighbors via online arithmetic," in *2022 IEEE 30th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2022, pp. 1–2.

[42] S. Gorgin, M. Gholamrezaei, D. Javaheri, and J.-A. Lee, "knn-msdf: A hardware accelerator for k-nearest neighbors using most significant digit first computation," in *2022 IEEE 35th International System-on-Chip Conference (SOCC)*, 2022, pp. 1–6.

[43] M. Alwani, H. Chen, M. Ferdman, and P. Milder, "Fused-layer cnn accelerators," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–12.

[44] J. Olivares, J. Hormigo, J. Villalba, I. Benavides, and E. L. Zapata, "Sad computation based on online arithmetic for motion estimation," *Microprocessors and Microsystems*, vol. 30, no. 5, pp. 250–258, 2006.

[45] T. Arifeen, S. Gorgin, M. H. Gholamrezaei, A. S. Hassan, M. D. Ercegovac, and J.-A. Lee, "Low latency and high throughput pipelined online adder for streaming inner product," *Journal of Signal Processing Systems*, pp. 1–15, 2023.