論文 / 著書情報
Article / Book Information

題目(和文)	アーキテクチャ－アルゴリズム協創による小型・高効率ニューラルネットワークアクセラレータの研究
Title(English)	A Study of Highly Compact and Efficient Neural Network Accelerators through Architecture/Algorithm Co-Exploration
著者(和文)	安藤洸太
Author(English)	Kota Ando
出典(和文)	学位:博士（工学）, 学位授与機関:東京工業大学, 報告番号:甲第11941号, 授与年月日:2021年3月26日, 学位の種別:課程博士, 審査員:本村 真人,髙橋 篤司,劉 載勲,中原 啓貴,原 祐子,佐々木 広,高前田 伸也
Citation(English)	Degree:Doctor (Engineering), Conferring organization: Tokyo Institute of Technology, Report number:甲第11941号, Conferred date:2021/3/26, Degree Type:Course doctor, Examiner:,,,,,,
学位種別(和文)	博士論文
Type(English)	Doctoral Thesis

Doctoral Thesis

A Study of Highly Compact and Efficient Neural Network Accelerators through Architecture/Algorithm Co-Exploration

Department of Information and Communications Engineering
Tokyo Institute of Technology

February, 2021

Kota Ando

2020 Abstract

A Study of Highly Compact and Efficient Neural Network Accelerators through Architecture/Algorithm Co-Exploration

Information and Communications Engineering

Kota Ando

Deep neural networks have enabled numerous AI applications in various fields, obtaining higher accuracy in many tasks than conventional heuristic approaches. This great success of deep neural networks was attained by advances in hardware technology typified by GPUs and the invention of model structure and training algorithms.

Dedicated hardware accelerators for neural networks on FPGAs and ASICs have been proposed to compute a neural network model in acceptable latency and throughput. On the other hand, the algorithms are continuously evolving, getting more complex and massive; hardware-oriented algorithms to reduce the computational complexity or amount have also been proposed. We expect that hardware-software coevolution drives further development of AI-enabled applications and establishes the next generation of information processing with AI-native computing power.

This thesis introduces programmable neural network processors for edge-side inference and accurate quantized neural network algorithms for embedded systems, with the concepts appropriately verified by FPGA and ASIC prototyping. Through this, we show a practical example and of hardware-software co-optimization, which proves its effectiveness by quantitative evaluation.

First, we prototype a reconfigurable parallel processor for convolutional neural networks. A convolutional neural network consists of two types of computational layers, convolution (Conv) and fully-connected (FC) layers, which have different computational and data structures. We extract a one-to-all parallel multiply-accumulation (MAC) as the common primitive operation in both Conv and FC layers. We construct an ideal parallel computational array with shared and individual buses to conduct this primitive operation. After that, we make this architecture feasible by relaxing internal data rates with an in-array data sharing mechanism and multithreaded accumulators. We discuss the requirements for hardware and model structures for better efficiency through the architecture exploration using this architecture.

We then propose a near-memory processor, named BRein Memory, which improves neural network processing efficiency by eliminating external data movements. It is known that the energy and latency of external memory access set limitations on the performance and efficiency of neural network processing. We attempt to omit the external memory throughout the entire processing of a neural network model. We adopt the binary neural network algorithm, where all the activation and weights are restricted to be -1/+1. It allows the multiplication to be calculated as bitwise XNOR. Thanks to this light-weight arithmetic, we integrate thin processing units between two SRAM macros to close the parallel computation and data movement in it. The computational units can be cascaded to compose a pipeline by matching the symmetric parallelisms in a neural network layer with the SRAM data access pattern, where no scratchpad memory is required. We fabricated a prototype LSI of this architecture with six processing blocks, which can house a 13-layer neural network model at most, achieving 2.3 TOPS/W energy efficiency, the best figure in neural network processor ASICs at the time published. The concept of highly efficient near-memory processing with a quantization algorithm has been proven by prototyping and evaluating this architecture.

Quantized neural networks reduce the computational complexity and memory footprint. However, the accuracy drop is unavoidable, and this could be a fatal problem in some situations. Here, we discuss an accurate yet efficient quantization algorithm that can be used in compact hardware implementation. A neural network processor is a digital signal processor when we see the computation carried in the processing units. The knowledge in signal processing gives us a hint for improving the accuracy of quantized neural networks on compact hardware. Dithering is a technique for low-precision quantization that reduces the quantization errors by representing the source data in a stochastically and spatially distributed manner. Its simplest algorithm is error diffusion, which can be computed by adding each pixel's quantization error to the neighbor. A processing unit in a neural network processor has an adder, which can be appropriated for quantization error accumulation. Based on this idea, we propose a quantization algorithm with error diffusion named Dither NN. Since we can use the adders that the neural network processor has, no additional arithmetic units are required for this extension. We implemented prototype architectures on an FPGA, and we trained convolutional neural network models with and without dithering. We proved the accuracy improvements with a very few additional hardware resource occupation through the evaluation.

This thesis provides a systematic methodology of architecture construction and a practical example of hardware-algorithm co-optimization, focusing on the computation primitives and data delivery patterns through these discussions and prototyping. The key

contributions of this thesis include 1) a reconfigurable architecture with coarse-grained data flow switching motivated by the potential common bases among the algorithms, 2) a proof-of-concept near-memory architecture enhanced by a quantization algorithm, and 3) algorithmic optimization of quantization reflecting the hardware structure to achieve higher accuracy without damaging the efficiency. The methodology extracting the essence of computational structure and data topology enables efficient and flexible processors, embodying a form of hardware-software coevolution.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The recent explosive advance of AI-related technology has brought the development of numerous applications in the smart society and ignited the research field of its mathematical basis and high-performance and high-efficiency computing. Among these AI methods, deep learning, or deep neural network, has been seen as one of the most promising techniques, by outperforming existing traditional AI methods (or even human) in various applications. Deep learning has experienced breakthroughs in this decade, which caused significant improvement in its accuracy.

Numerous variations of the deep neural network are proposed almost daily. Training techniques to achieve higher accuracy in a short development time are also researched. Their supporting computational resources are getting richer, including GPU-powered data centers, personal workstations, and edge-side/embedded systems. These researches and developments have realized AI-based applications in various domains.

1.1 Motivation

The historic win of AlexNet in ILSVRC 2012, an image recognition contest, made the research of deep neural networks active and attractive. Since that, countless papers on deep neural networks have been presented almost every day.

Neural network algorithms are continually evolving. The progress of research and development of neural network models has brought significant accuracy improvements in various fields. However, the amount and complexity of computation have also increased along with it.

Neural networks' potential ability has elaborated various applications, including mobile, car-mounted, and IoT edge uses. It is desired to process the neural-network-based applications with short delay yet low power consumption on such devices. There have been

some new requirements, such as privacy, for which server-side processing is not suitable. Thus, we need to process neural networks on those low-power devices with an acceptable throughput and latency. Optimized hardware is the hopeful solution for the efficient and feasible adoption of neural network technology in edge devices. In this observation, we expect the coevolution of neural network algorithms and programmable processors to propel advances in AI-powered smart edge devices.

1.1.1 Neural Network Processors

The great success of neural networks cannot be told without the development of hardware technology, especially GPUs. The principle of the neural network was invented in the 1950s [70], the basis of backpropagation for training multi-layer models was disclosed in 1986 [71], and the first modern convolutional neural network was proposed in 1998 [44]. However, training a large neural network model was impossible until the GPU-powered parallel computation became available around 2010. AlexNet, the first dark horse of deep neural networks, came in 2012, supported by the two-GPU training environment [43].

Today, deep neural networks have become a major application of GPUs. The latest-generation GPUs have adopted a high-performance matrix calculation unit called Tensor Core, mainly targetting deep neural network training and inference.

Dedicated neural network processors on FPGAs and ASICs have been actively researched too. These processors' main objectives are throughput and power efficiency, but programmability for accomodating various types of models has been emphasized.

1.1.2 Neural Network Algorithms

After AlexNet's reputation, numerous trials for convolutional neural networks are conducted. They tended to have more and more layers and channels to achieve higher accuracy, which caused an incredible explosion of computational and memory demands in exchange for new-record accuracy. In 2014, the VGG-16 model [76] achieved > 70% accuracy in ImageNet classification, which is about a 15-point better result than AlexNet, with about ten times more operations. Following it, ResNet [32] successfully improved the accuracy with a huge number of layers but fewer parameters by introducing the residual skips. The residual skips and newly proposed batch normalization [37] were breakthroughs that allow massively deep networks to be stably trained. MobileNet models [34, 72] achieved a comparable accuracy with much fewer parameters by decomposing the basic convolution operations. Pruning and model compression are other techniques to reduce the memory footprint with retaining accuracy.

In parallel to these improvements of model structures, many hardware-oriented algorithms were proposed. Quantization is the essential technique that allows us to train and deploy the network model to a simple and small hardware structure [83, 20, 68, 18, 53, 45, 81, 21, 9, 40].

The model algorithms are continuously evolving. For the realization of these emerging algorithms to practical applications, hardware supports are necessary. Thus, we anticipate the coevolution of model algorithms and programmable processors.

We have FPGAs and ASICs as means of this purpose. An FPGA can offer desired hardware implementation with a shorter development time and lower initial cost, enabling a try-and-error verification and optimization. On the other hand, ASIC development is generally expensive but promises the best efficiency, and its freedom of design could bring a novel paradigm of computation. Here, we aim at the generation of a neural-network-native efficient and flexible platform from both sides of software and hardware, with the help of our weapons of FPGAs and ASICs.

1.1.3 Objectives of This Thesis

Convolutional neural networks have become the mainstream of AI algorithms since AlexNet won the image recognition competition in 2012 [43]. A convolutional neural network employs convolutional (Conv) layers to extract 2-dimensional features from their inputs by applying discrete convolution. For image recognition (classification) purposes, traditional perceptron-like classifier layers called fully-connected (FC) layers follow the Conv layers. The key idea of convolutional neural networks is that trainable Conv layers obtain the optimal feature extraction without heuristics [44]. The computational procedure difference between Conv and FC layers makes it difficult to build a universal processor for convolutional neural networks. Though ASIC- or FPGA-based convolutional neural network processors have been proposed, they tend to concentrate on Conv layers and ignore FC layers. Conv layers usually dominate over FC layers in orders of magnitudes at the viewpoint of processing time; hence accelerating only Conv layers is a reasonable approach. However, FC layers are still indispensable building blocks for classifier purposes [39], so the absence of FC layer support results in a limitation of the entire system. Balancing the performance and efficiency of Conv and FC layer processing is necessary when designing a versatile neural network accelerator. In that sense, we conduct a comprehensive architecture exploration seeing the performance and efficiency of both the Conv and FC layers.

It is known that Conv layers are computation-bound; the performance is restricted mainly

by the number of operations [91, 86]. In contrast, FC layers are memory-bound, where the memory bandwidth is the primary hardware limit. Also, the energy consumed for data transfer from external memory is much larger than the computation [12]. This fact motivates us to consider omitting the expensive external data accesses in FC layers. To eliminate the energy-hungry external memory, we must use an internal memory commonly consisting of SRAMs both for scratchpad memory and parameter (*i.e.*, trained weights) memory. If the whole processing is completed nearby the SRAMs, no data transfer is required; this is the basic idea of near-memory (or in-memory) computing. The internal SRAM buses can generally have broader bandwidth than external buses; it is expected that highly parallelized computing can be realized with this idea.

On reflection, a neural network layer potentially involves various opportunities for parallel computation. Putting the SRAM's bit-wise parallelism together, we can exploit the potential parallelism in a neural network layer. This enables a near-memory neural network processor that requires no external data access, which improves the efficiency of FC-dominant neural network models. We also adopt a neural network approximation technique called quantization to compress the data amount. Quantization allows relatively small SRAMs to store entire weights and compact processing units to be parallelized near SRAM macros.

Quantization is a widely used technique for simplifying the computation and reducing the required data amount, thus improving the performance and efficiency of neural network processors. It brings a trade-off between efficiency and accuracy; strict quantization may result in significant accuracy degradation. Efficiency and performance are the main issues of architecture exploration, as mentioned above, but accuracy is also an essential criterion of a neural network application. An efficient and accurate quantization that be processed in compact processing units and retain a higher accuracy is desired.

A neural network processor can be seen as doing digital discrete signal processing. Data quantization has also constituted a major theme in the field of signal processing. Knowledge in signal processing gives us a hint to a novel accurate quantized neural network algorithm when observing the data transformation handled in neural network hardware. We then propose a hardware-native and neural-network-native quantization algorithm that improves accuracy while maintaining quantized neural network processors' efficiency and compactness.

1.2 Contribution of the Thesis

The main contributions of this thesis include the following.

- A systematic and practical example of neural network architecture exploration is provided considering the primitive operations and data movements.
- A concept of efficient near-memory neural network processing is proved, and its LSI implementation and algorithmic optimization are discussed.
- A hardware-oriented quantization algorithm that improves accuracy without degrading quantized neural network architecture's high-efficiency nature is presented.

The rest of this thesis is composed as follows.

First, in Chapter 2, we look around the research topics of AI hardware. Contributions of prior researches to the AI architectures and hardware-aware algorithms are briefly introduced. Explanations of related work are detailed in each of the later chapters.

Chapter 3 explores a simple parallel architecture for typical workloads of deep convolutional neural network models, which provides prerequisites and criteria for more specialized discussions. We first look at the two dominant types of neural network layers: convolution and fully-connected layers. After a prototype architecture is presented, we eventually modify it to a practical one. We discuss the optimal computational units, control, and memory structure by observing their arithmetic and data access patterns. These observations and discussions constitute the motivations and criteria of architectures and algorithms presented in later chapters.

In Chapter 4, a near-memory parallel architecture named BRein Memory is presented. According to Chapter 3, the continuous internal/external memory accesses affect the energy and processing time of the entire neural network processing; here, we evaluate an extreme case that all the memory access and processing are completed near the internal memories. We construct a near-memory fully-connected neural network processor architecture based on this motivation, fabricate its prototype LSI, and evaluate it. We adopt the binary neural network algorithm to simplify the computation and reduce the memory footprint to enable such extreme data flow and computation.

In Chapter 5, we attempt to improve the accuracy of hardware-friendly quantized neural network models focusing on the algorithm's signal processing counterpart. Chapter 4 proves the concept of near-memory processing with the quantization algorithm for high-efficiency embedded systems, but the accuracy drawback becomes a new problem. Inspired by a signal processing viewpoint of neural network processor architecture, we then improve the accuracy of quantized neural network models without significant efficiency degradation.

Finally, Chapter 6 concludes the thesis with the outlook for future enhancement of AI applications, which is prospected to be driven by looking broader and deeper into the

nature of algorithms and hardware.

Chapter 2

Background

The research topics on neural network algorithms and architectures are discussed actively and changing continually. In this chapter, we go on an excursion to the background and history of neural network technology, introducing the latest contributions as well as monumental achievements.

2.1 The Arithmetic of Neural Networks

The task of a neural network model is simply the transformation from its input to the output by a trainable set of parameters (weights). Multiply-accumulation (MAC) operations dominate the computation in a neural network model. Since the systematic MAC operations between the weights and inputs can be denoted as a matrix-vector product, matrix processing accelerators like GPUs have been used for neural network computation. The rest of the neural network model processing contains non-linear activation functions, pooling operations, and shape manipulation (e.g., vector concatenation). In this section, we take a look at the arithmetic used in neural network processing.

2.1.1 Convolution and Fully-Connected Layers

A typical convolutional neural network model employs several convolution (Conv) layers, followed by fully-connected (FC) layers. A Conv layer applies 2-dimensional convolution to the input to extract spatial features (patterns). An FC layer conducts a linear transformation of the input, similar to a classical perceptron. Typically both types of layers are followed by a non-linear function (called activation function). It is understood that Conv layers act as trainable feature extractors in a classification use, and FC layers form a classifier [44].

There are convolutional neural network models without FC layers, e.g., for non-classification applications, such as object detection [69], segmentation[84], generative models [66], and image super-resolution [87]. Delated convolution (also called deconvolution or transposed convolution) used in these models is a special case of convolution.

2.1.2 Batch Normalization

Batch normalization (BatchNorm) [37] appeared in 2015 and immediately became an indispensable component in neural networks. It statically normalizes the distribution of the pre-activation sum in a neural network layer as follows:

$$\hat{u}_{i,j,c} = \gamma_c \frac{u_{i,j,c} - \mu_c}{\sigma_c} + \beta_c \tag{2.1.1}$$

Here, $u_{i,j,c}$ is the MACed sum at the pixel (i, j) in channel c. μ_c and σ_c are the mean and standard deviation of the channel c, while β_c and γ_c are the bias and weight affine parameters, respectively. The part $\left(u_{i,j,c} - \mu_c\right)/\sigma_c$ in Eq. 2.1.1 is a well-known statistic standardization assuming a Gaussian distribution. The key idea of batch normalization is that the mean μ_c and standard deviation σ_c are calculated in the batches in the training phase and accompany the affine parameters β_c and γ_c. The affine parameters β_c and γ_c are also trainable during the backpropagation-based standard training process, which absorbs the power disparity among channels.

Batch normalization enabled the neural network models to be trained in a shorter time with good stability away from gradient vanishing or divergence. Later described quantization techniques strongly rely on the distribution adjustment functionary of batch normalization.

2.1.3 Skip Connection

Skip connection, or residual skip, is a bypass path in a block with no computation and is added to the mainstream path's output, proposed in [32]. Reference [32] successfully trained a very deep convolutional neural network, named ResNet-152, consisting of 152 computational layers, achieving a state-of-the-art recognition accuracy. The skip path retains and carries the gradient with fewer intermediate layers, which prevents the gradient vanishing problem even in these massively deep network models.

Today, many models utilize the skip connections in some way. MobileNetV2 [72] invented a 3-layer building block structure called an inverted residual block, which locally expands the channels to obtain a better information extraction.

2.1.4 Separable Convolution

Separable convolution is a computation- and memory-reduction algorithm of convolution, devised in the MobileNet [34] proposal inspired by [74] Here, let us understand the core ideas and building blocks of MobileNet.

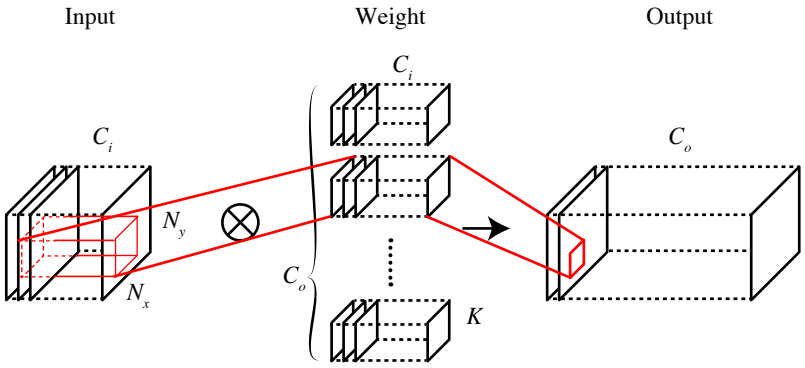

Fig. 2.1: Computation of standard convolution.

A standard convolution (Fig. 2.1) calculates element-wise MACs of fully-connected channels costing $K^2C_iC_oN_yN_x$ operations and $K^2C_iC_o$ parameters for $(K \times K)$-sized kernel, C_i input / C_o output channels, and $(N_y \times N_x)$-shaped output. When the channels are divided into G groups, the number of operations is reduced by $1/G$ because each output channel relies only on the inputs in the same group, as shown in Fig. 2.2. This is called *grouped convolution*.

Assume $G = C_i = C_o$; each output channel is produced using only one input channel, requiring only $K^2C_oN_yN_x$ operations for the entire output channels, as depicted in Fig. 2.3. Here, each output channel represents the transformed (or extracted) feature from a single input channel, in contrast to an output of the standard Conv extracting the feature among all the input channels. This operation is called *depth-wise convolution* since the operation is closed within each depth (channel).

Then, assume $K = 1$; only one weight exists in each channel. The convolution of an output channel uses each input pixel only once; it no longer considers the correlation

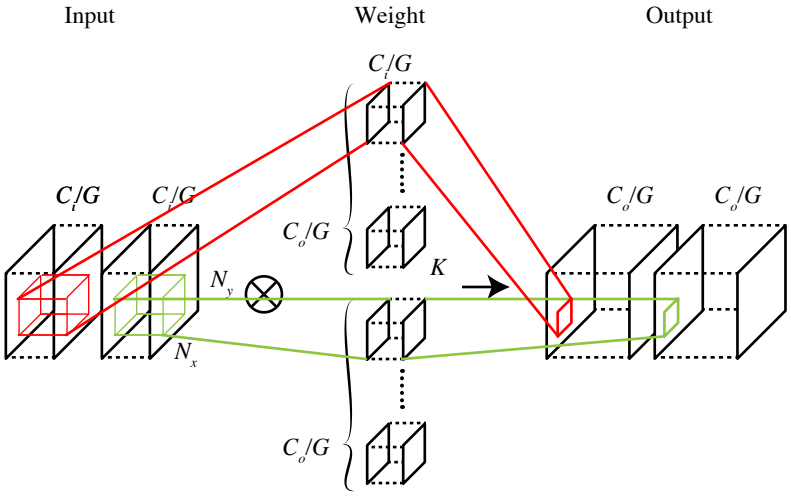

Fig. 2.2: Computation of grouped convolution ($G = 2$ in figure).

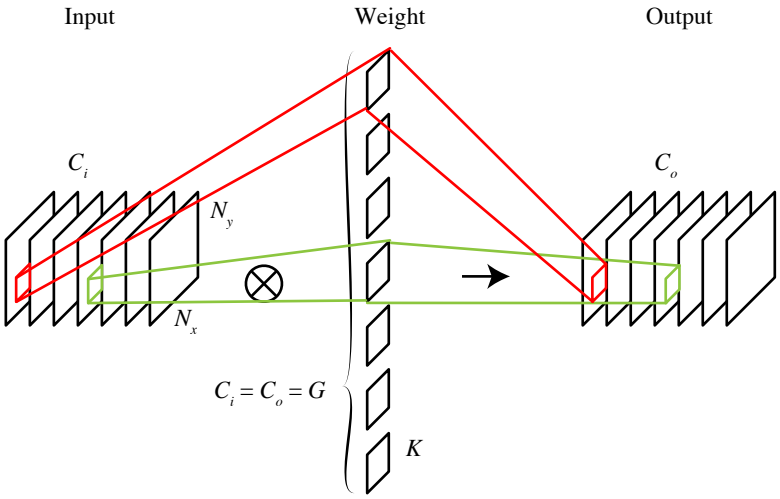

Fig. 2.3: Computation of depth-wise convolution.

between adjacent input pixels, *i.e.*, it extracts only the relationship between input channels. This operation is done for each pixel independently; it is called *point-wise convolution*.

MobileNet employs separable convolution blocks as its basic building blocks, decomposing a Conv layer into the sequence of depth-wise and point-wise convolution layers.

Remember that a Conv output channel extracts a spatial feature laying among the input channels. The essence of separable convolution is to divide this task. The spatial information is transformed in the depth-wise convolution while the correlations between channels are treated by the point-wise convolution. By separating the point-wise and depth-wise convolution, the computation amount is relaxed from $K^2C_iC_oN_yN_x$ to $\left(K^2 + C_i\right)C_oN_yN_x$, while the number of parameters is reduced from $K^2C_iC_o$ to $\left(K^2 + C_i\right)C_o$. MobileNet accomplished a comparable accuracy to VGG-16 [76] in ImageNet classification with 26× and 32× less computation and memory, respectively. Today, its extended versions, MobileNetV2 [72] and MobileNetV3 [33], have been proposed to achieve further computational load reduction and accuracy improvement in a variety of target tasks.

2.2 Hardware for Neural Network Processing

As neural networks have attracted attention, many types of hardware accelerators have been developed in the academy and industry [83]. They can be categorized by several properties. First is the accelerator's purpose, that is, the accelerator designed for training or inference. The dominant operations in a neural network in the training and inference phases are multiply-accumulation (MAC operations), but the processing flow and memory usage are different. Second is the implementation/fabrication technology on which the accelerator relies. FPGAs and ASICs are used as the implementation platform for neural network accelerators. In this section, we take a brief look at these neural network hardware implementations.

2.2.1 FPGA-based Implementation

FPGA is often used for edge-side and server-side neural network inference and training [27, 90, 58, 38]. The advantage of FPGA is its flexibility, where neural network model structures vary to match their target applications, and the model algorithms continue evolving. It is also possible to elaborate a hardware implementation that is fully optimized/customized for a specific network model and application in the user environment. SoC-enabled FPGAs that can invoke the developed coprocessor from a Linux program have made practical application development on an FPGA much more convenient.

Reference [91] analyzed the performance and efficiency of FPGA implementation using the roofline model illustrated in Fig. 2.4. It plots the arithmetic intensity (operations per byte; the number of operations that can be delivered using data loaded once) in its X-axis and computation performance (operations per second) in its Y-axis. The reachable

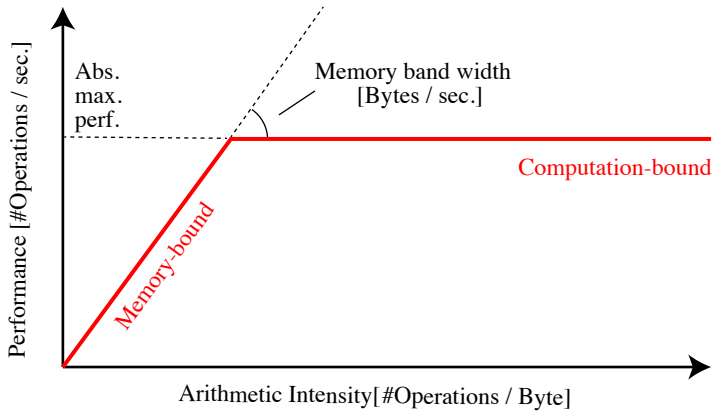

Fig. 2.4: Roofline model.

performance under a hardware configuration is constrained by memory bandwidth and theoretical maximum computation performance, whichever lower. This model suggests that when the arithmetic intensity is lower, *i.e.*, when data reuse is insufficient, higher performance cannot be achieved because of idle time waiting for the memory data movement. They conducted comprehensive architecture exploration using this model, trying various architecture optimization methods such as loop unrolling and pipelining, providing a quantitative methodology of architecture optimization and evaluation.

The advantage of using an FPGA is that an end-user can program and customize the hardware implementation easily and quickly. Hence, a user can have the FPGA co-optimized with their original neural network models and applications. However, hardware development and optimization are not so easy for software developers. Automated design exploration and parametrized design compilers are proposed, which reduce the difficulty for software developers of using their own neural network models on FPGA-based accelerators [80]. GUINNESS [59] is an open-source framework for constructing, training, and deploying a neural network model to an FPGA. The FPGA design is automatically generated, for which the user does not have to be experienced in hardware development. ONNC [50] is an open-source hardware compiler compatible with the ONNX format. There are also proposals on hardware-specific model exploration not limiting for FPGAs [78, 10].

2.2.2 Neural Networks on ASIC

ASIC is another solution for neural network acceleration. Despite the flexibility limitation and long development time, the motivations to integrate a neural network processor into an ASIC against an FPGA include:

- A higher degree of freedom in the design. There are no restrictions in hardware design topics such as logic/arithmetic structure and data transfer.
- Higher efficiency. An FPGA implementation *emulates* the architecture using its predefined structure, not directly *realizes* the structure.

As for design freedom, even the physical specification cloud be methods for efficiency improvement; *e.g.*, Envision [55] attained further power saving by dynamic power supply voltage control. Optimizing neural network models for a specific hardware architecture may satisfy the accuracy, power, and performance requirement in an easier method than constructing a highly optimized architecture for an existing network model on an FPGA.

Here, we tour some notable achievements of prior ASIC implementations for neural network processing.

DianNao series accelerators were one of the earliest neural network inference accelerators [11, 14, 15, 23]. Eyeriss was another pioneering convolutional neural network accelerator ASIC, which featured efficient data reuse on NoC [17, 12, 13]. Their work was also remarkable in that they categorized the data delivery patterns into several types of *stationary* and analyzed the hardware costs of each stationary [13, 77]. Reflecting the trend of sparse neural network models, they later expanded this NoC-based dataflow controlling in Eyeriss v2 [16].

DNPU [73] was one of the accelerator LSI that can process recurrent neural networks (FC-centric models) in addition to convolutional networks by having two different (heterogeneous) computation unit arrays. Its successor UNPU [47, 46] unified the computation resources for Conv and FC layers by introducing an internal data sharing mechanism and supported variable bit-width fixed-point arithmetic for flexible accuracy/efficiency optimization. LNPU [48] was a DNPU-family neural network processor with on-chip training support powered by low-bit-width floating-point arithmetic. GANPU [41] followed it targeting single-chip GAN processing (generative adversarial networks) with reconfigurable task mapping on NoC. The transition of the philosophy of parallel computation and its primary target tasks through this series of proposals implies the trends of embedded/edge-side neural network applications.

Commercial neural network accelerators have been in operation. Google's TPU is a server-side accelerator that can be used for training and inference, which is available for

their cloud service users. They also released Edge TPU, a low-price consumer-use product of neural network accelerator, which can be invoked from a host PC via USB.

2.3 Trends of Neural Network Algorithms

2.3.1 Quantization

Quantization, *i.e.*, low-precision numerical expression, is one of the most commonly used techniques when mapping neural network models onto hardware.

In most mathematical computation on a CPU or GPU, the arithmetic is almost always in floating-point expression. 64-bit and 32-bit floating-point expressions (FP64 and FP32) are available in modern CPUs. It is preferred for various computation purposes, including the training phase of a neural network model, because the floating-point number format can represent a broad range by its exponent factor. For example, the maximum value of an FP32 number is $\approx 2^{128}$ in 32 bits, while a 32-bit fixed-point number can range from -2^{31} to $\left(2^{31}-1\right)$. However, the computation of floating-point numbers is complicated. It does not become a problem when we use a CPU because modern CPUs are commonly equipped with FPUs (floating-point processing units), but the situation may differ in low-power embedded systems. Generally, floating-point numbers need relatively complex logic for calculating mantissa with regulating the exponents and signs.

So, many neural network processors are built with quantization methods to avoid using fixed-point arithmetic. A commonly used one is the fixed-point expression. A fixed-point number is an integer in its treatment on a digital circuit, *i.e.*, it consumes as much power and hardware resources as integer arithmetic, not requiring any additional logic units. The bit width of a fixed-point number is also a topic in architecture exploration. Neural network configurations with 16-bit, 8-bit, 4-bit, and more types of fixed-point expression are reported [83]. We should note that we cannot estimate the numerical range (*i.e.*, distribution) of the activations and weights in the early training phase of a neural network model, so we should use a floating-point expression. Quantization algorithms adjusting the scaling according to the dynamic range of the data are called dynamic quantization [19, 88].

Neural network models using 1-bit binary quantization (or simply *binarization*) are called binary neural networks [20, 68, 18]. It is equivalent to the 1-bit fixed-point numerical expression, where only the sign bit remains. It can drastically simplify the computation by replacing arithmetic multiplication with a simple bit-wise operation. Logarithmic quantization is also utilized for low-power neural network implementation [53, 45, 81].

It also omits costly multiplication by representing it as an arithmetic addition. Thus, logarithmic quantization can realize multiplication operation in neural network processing in a better approximation than the binary quantization. These extreme approximation methods often suffer from the difficulty of parameter exploration and accuracy degradation but can be affordable choices to reduce the hardware resource and power consumption.

We introduce a highly optimized accelerator architecture by adopting a binary algorithm in Chapter 4. We also propose a quantization algorithm that exploits the nature of the procedure of quantized neural network processing to improve accuracy with minimum hardware cost in Chapter 5.

As mentioned above, the numerical ranges in a neural network model can hardly be predicted in the phase of model construction, making it difficult to determine the optimal hyperparameters in hardware implementation, such as bit width in fixed-point quantization. Instead of fixed-point representation, floating-point systems with reduced bit widths have been utilized both in GPU training and edge-side inference [85, 64]. It is reported that reduced floating-point expression, *e.g.*, FP16 and bfloat16 [36], can obtain better accuracy without hyperparameter exploration (which is often severe with fixed-point expression) [21, 9, 40]. Although the hardware realization of floating-point arithmetic is costly than fixed-point one, it would be an acceptable trade-off point of easiness of model construction and processing efficiency. The latest generation of GPU natively supports FP16 and bfloat16 computation, and a commercial accelerator for cloud training using bfloat16 has been disclosed [64, 85].

2.3.2 Model Compression Techniques

Pruning is a fundamental technique to reduce the neurons or synapses (*i.e.*, the dimensions of activation vectors or weight matrice). During the training phase, neurons or synapses that are regarded as ineffective are removed. This can reduce the required number of operations and memory footprint, thus relax the hardware resource claims.

The simplest algorithm is to add a regularization term in the loss function and force the weights having values below the threshold to be zero [31]. In the inference phase, we need to skip the calculation of pruned synapses or neurons to benefit efficiency/performance gain from pruning; therefore, the information on locations of pruned weights must be held. Deep Compression [30] introduced an iterative strategy to achieve the desired pruning ratio and quantization configuration. It also proposed a weight sharing algorithm with Huffman-code-based weight indices (*i.e.*, the locations of living synapses). It was then integrated as a hardware accelerator named EIE [29].

2.3.3 Hardware-aware Model Algorithms

A binary neural network algorithm assuming itself mapped on an FPGA was proposed [25]; it was later published as an open-source project named BinaryBrain. An FPGA replicates digital circuitry by forming a functionally equivalent circuit using look-up tables (LUTs) and switch blocks. The number of inputs and outputs of a LUT is limited (*e.g.*, 6-bit input and 1-bit output in Xilinx 7 series products). Hence, if each neuron in a binary neural network layer is forced to have at most six inputs, the FPGA can house the model the most efficiently, where each neuron can be seated on a single LUT. BinaryBrain trains the model with a time-domain stochastic numerical expression to gain higher accuracy under this 6-to-1 neuron precondition. This time-domain expression is differentiable; thus, it can be trained by standard backpropagation. LUTNet [82] is another FPGA-native neural network implementation, which interprets a LUT as a binary activation function that can take arbitrary mapping from the input to output bits.

Chapter 3

Architecure Exploration Focusing on the Diversity of Convolutional Neural Network Processing

3.1 Introduction

A convolutional neural network model consists of several types of layers. The main two types of them are convolutional (Conv) layers and fully-connected (FC) layers, both of which employ multiply-accumulation (MAC) operation between the activation and weight. A Conv layer operates feature extraction of the 2-dimensional input, while an FC layer computes linear transformation of its unstructured input activation.

Most convolutional neural network processors have made their effort to process Conv layers efficiently since the Conv layers account for a significant part of the processing time. However, FC layers are still indispensable in many applications. Therefore, the lack of support for FC layers limits the availability of the architecture. In this chapter, we conduct architectural exploration to present a feasible form of a neural network processor capable of computing both the Conv and FC layers with a single architecture.

The rest of this chapter is organized as follows. First, we take a look at the computational structures of Conv and FC layers in Section 3.2, focusing on the similarity and difference of their arithmetic and memory access patterns. Second, an ideal base architecture from which we start the architectural discussion is presented in Section 3.3. Third, we modify the architecture into a feasible one considering the hardware resources, memory bandwidth, and processing time in Section 3.4. We then evaluate those architectures assuming typical workloads in realistic neural network models in Section 3.5. Finally, we conclude the discussion of this chapter in Section 3.6.

This chapter is based on previously presented conference and journal papers [2] [4].

3.2 Computation and Data Access of Convolutional Neural Networks

As mentioned above, a convolutional neural network mainly consists of several Conv layers followed by FC layers. A Conv layer computes 2-dimensional convolution between its 2-dimensional input activation (an image or feature map with multiple channels, representing the input "feature") and the corresponding weight (kernel) for each output channel, which extracts the 2-dimensional feature resembling the weight pattern from the input. An FC layer has the same structure as a classical perceptron, *i.e.* , it transforms the 1-dimensional input, which acts as the classifier. Although these layers have different structures of 2- and 1-dimensional inputs and weights, the basic computation is MAC operation in both of them. In this section, we look at the similarity of the Conv and FC processing from the viewpoint of data transfer.

An output pixel of the Conv layer is calculated using the kernel weights shaped $K \times K$ and the partial area (window) of the input activation with the same shape; the value is the sum of the pixel-wise product (MAC operation), as Fig. 3.1(a) indicates. The next output pixel is calculated similarly, with the same weights and the input area slid by several pixels. This interval of input windows is called stride s. A Conv layer can be seen as a special case of an FC layer, in which each output neuron has the connections only to a limited set of the input neurons, and the weights are shared among output neurons, as Fig. 3.1(b) depicts.

Since all the output pixels share the same weight values, each pixel in an output channel is calculated using all the kernel weight values corresponding to the channel. On the other hand, when we focus on an output pixel and a weight value, the input area location to be MACed is uniquely determined. Therefore, the convolution can be computed by selecting a kernel weight value one by one and multiplying the corresponding input area for all output elements, as indicated in Fig. 3.1(c).

From another view, as shown as solid black lines in Fig. 3.2, an input element influences all output channels. At the same time, a weight value is shared among all the output and input pixels, as indicated as broken lines in Fig. 3.2. In a Conv layer, we can observe the data reusability in both inputs and weights.

An FC layer is calculated as a matrix-vector product of the weight and input, as shown in Fig. 3.3(a); thus, an output value can be denoted as a dot product of the row vector in the

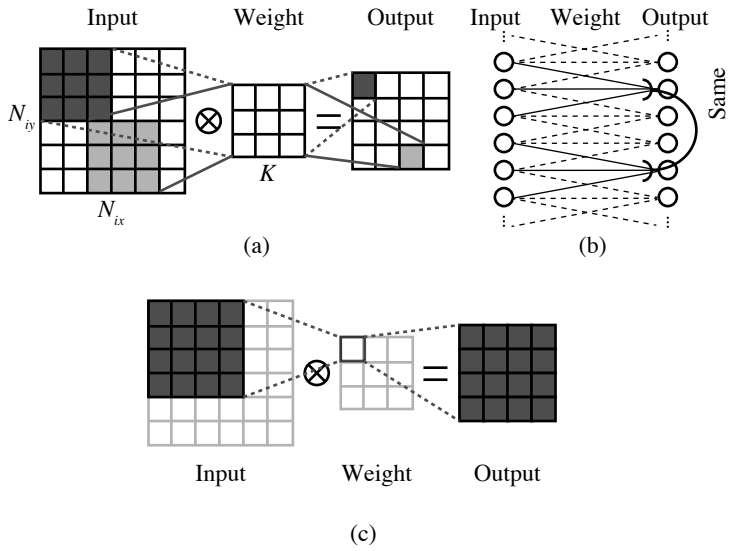

Fig. 3.1: The structure of a Conv layer (a)(b) and the data dependency(c).

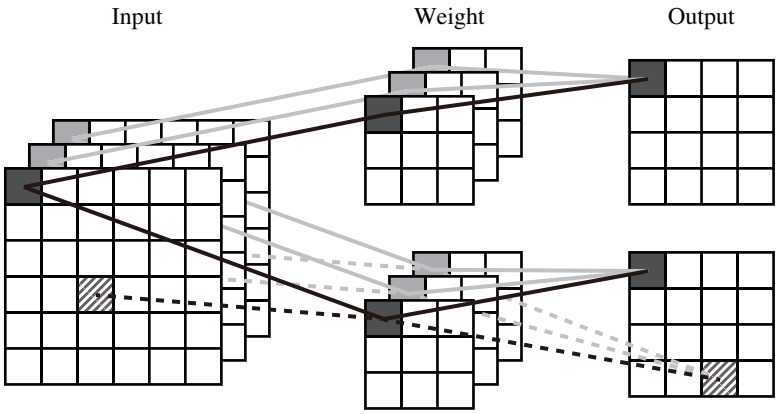

Fig. 3.2: Data reusability in a Conv layer.

weight matrix and the input vector. Unlike a Conv layer, the weight values in an FC layer are never shared, as depicted in Fig. 3.3(b). Therefore, the only opportunity for data reuse in an FC layer is input reuse, where an input value is reused for all the output elements. In other words, an element of the input vector is (and can only be) used once per weight row vector. The computation of an FC layer can be processed by scanning the input value (*i.e.*, an element of the input vector) cycle by cycle, conducting a MAC operation with the

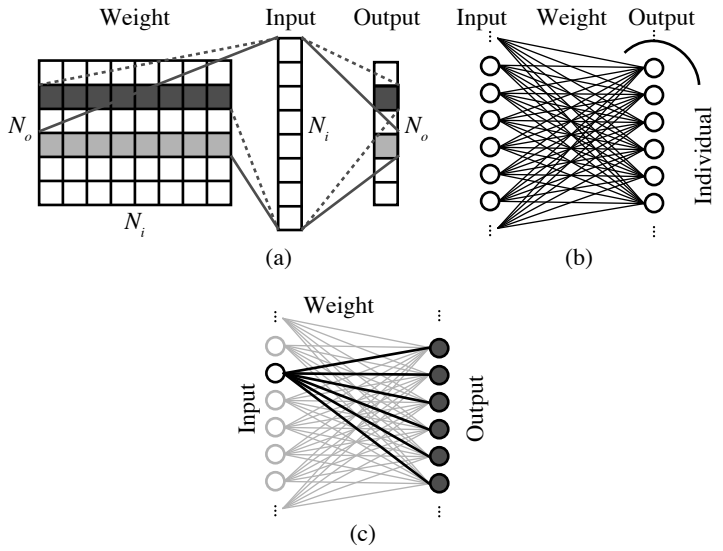

Fig. 3.3: The structure of an FC layer (a)(b) and the data dependency(c).

weights for all the output neurons (a weight row vector), as indicated in Fig. 3.3(c).

3.3 Multithreaded CGRA Architecture for Input Value Reuse

The previous section showed that both Conv and FC layers employ MAC operations as the basic computation and that the data sizes of input and weight are reversed. This section presents an ideal parallel processor architecture that can efficiently process both the Conv and FC layers with a single structure by switching only the data flow. This architecture assumes a very high internal bandwidth (between buffers and processing elements), which is hardly realistic. We then modify it into acceptable and realistic structures later in Section 3.4.

3.3.1 Base Architecture

Here, we construct a simple parallel processor for MAC operations by an array of the multiply-accumulator-based processing elements (PE). A PE computes a pixel in a channel of the output feature map of a Conv layer and an element of the output vector of an FC layer.

This type of parallel processing is called "output-parallel" since each PE corresponds to an output element, and the entire array produces multiple output values. In a temporal view, this architecture is also an "output-stationary" one, as the output values being computed remain for several cycles while the input and weight are fed cycle by cycle.

In the processing of a Conv layer, one weight value is used among multiple output pixels in a channel. On the other hand, in an FC layer processing, even though there is less opportunity to reuse data compared to a Conv layer, one input value is used for the computation of all the output elements. This observation leads to the idea of an output-parallel PE array with a "shared" input bus that broadcasts a single value to the entire array in each cycle and an "individual" bus that feeds a value to each PE.

Fig. 3.4 shows the data delivery in a Conv layer. Typically a Conv layer has multiple input and output channels, but here we show a case where both the input and output activations have only one channel for simplicity. Again, the PE array processes the output plane, where each PE corresponds to an output pixel. Here, the shared bus picks a kernel element in a cycle and feeds it to all the PEs. The individual bus crops the input activation area used by the output through the kernel element currently picked on the shared bus. Fig. 3.6 indicates the procedure of this processing. We define the PE array size (parallelism; the number of PEs in a row/column) as $P_x \times P_y$, the kernel size as $K \times K$, and the stride of the Conv operation as s. This example is for the case of $P_x = P_y = 4$, $K = 3$, and $s = 1$. For each cycle, one pixel in the kernel is broadcasted in the PE array via the shared bus, and the corresponding input area sized $P_x \times P_y$ is cropped and fed via the individual bus. This process is repeated for all $K \times K$ kernel elements. When the output size exceeds the PE array size, time-division processing is required.

Fig. 3.5 presents the computation of an FC layer. For the FC layer processing, one input value is used by multiple output values, in contrast to the Conv layer that reuses a single kernel weight value. An input value and multiple weight values are fed in a cycle via the shared and individual buses, respectively, to conduct a parallel MAC operation. The weight values are from a row of the weight matrix, which corresponds to weights from an input activation to multiple output neurons, without respect to the 2-dimensional manner of the PE array. The procedure of an FC layer processing is indicated in Fig. 3.7. Here, we define the input size (*i.e.*, input vector size; the number of input neurons) as N_i and the output size as N_o. The weight matrix is shaped $(N_i \times N_o)$. Each PE produces an output activation; the entire array computes a parallel MAC operation from an input to $P_x P_y$ output activations (neurons) in a cycle and repeats for all the input values sequentially. If the output size No is greater than the PE array size $P_x P_y$, the computation is divided in a time-division manner.

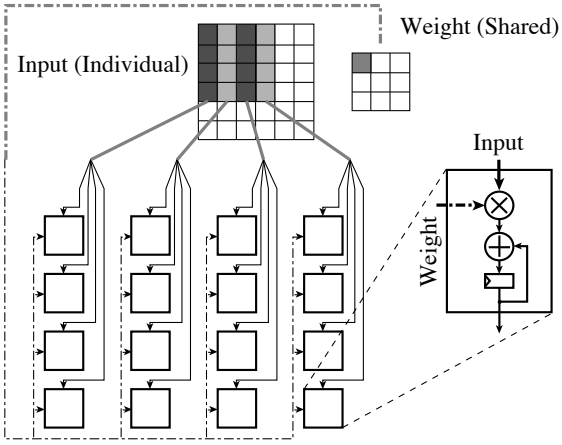

Fig. 3.4: Data delivery in a Conv layer.

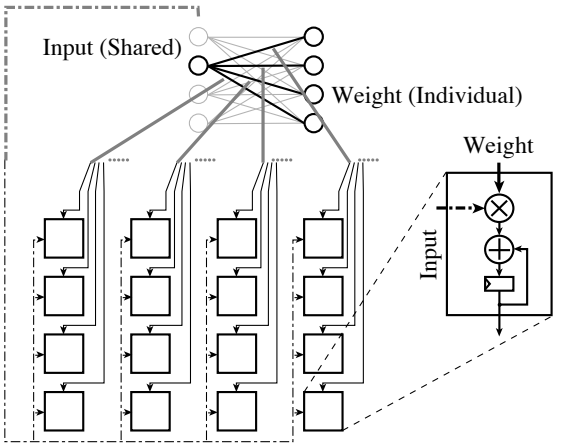

Fig. 3.5: Data delivery in an FC layer.

3.3.2 Multithreaded Accumulator for Data Reusing

We have discussed the basic ideas in an ideal case, where the attainable bandwidth of the individual bus is very high. To construct a more feasible architecture, we should consider lowering the required data rate. In general, a smaller amount of data transfer per operation is preferred most from the viewpoint of higher power efficiency (the reciprocal of this

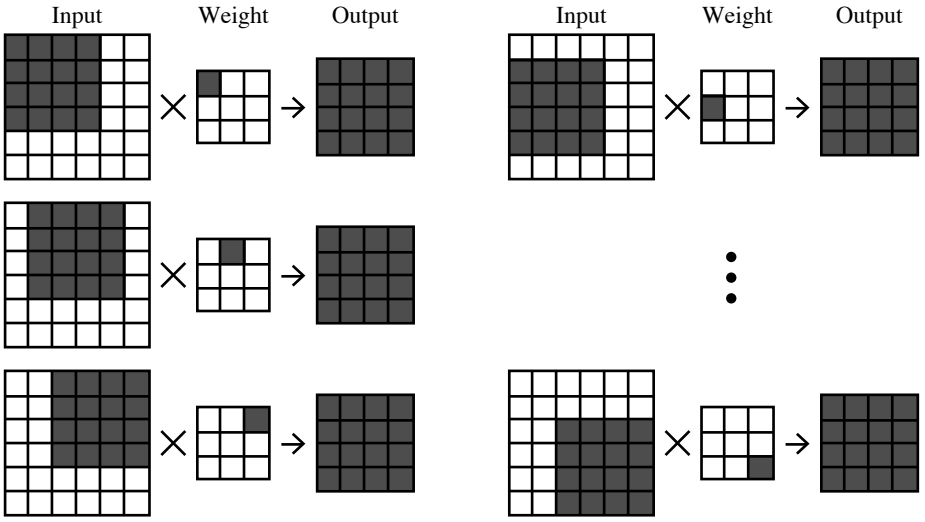

Fig. 3.6: Basic procedure of Conv layer processing.

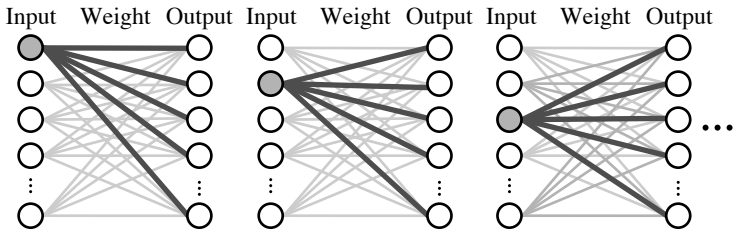

Fig. 3.7: Basic procedure of FC layer processing.

measure, the number of operations that can be conducted using a certain amount of data transfer, is called arithmetic intensity, usually in the unit [OPs/Byte]). We present two ideas that increase arithmetic intensity in this section, and we discuss modifying the ideal architecture into a feasible one in Section 3.4.

First is the sequential locality of the partial input area of Conv layers. Since the $K \times K$ kernel elements of an input channel are scanned sequentially, and since the K elements in each row are consecutive when the stride is 1, there are chances of input data reuse in adjoining partial input areas in each kernel row. Therefore, the partial inputs referenced by the next weight are the same except for the last column. In the same weight row, the input elements required by the PEs are the input values that the adjoining PEs loaded in the previous cycle. By reusing these already-loaded input values in a kernel row, the data

transfer amount can be decreased to at most $1/K$. Fig. 3.8 depicts this idea, where the input pixels hatched in gray is the pixels to be reused.

In second, there can be observed the temporal locality of the input in the multiple channel convolution. Let the numbers of input and output channels C_i and C_o, respectively. One partial input area is used C_o times because each output channel depends on all input channels. Once the input values of a partial area are fetched to calculate the first output channel, we can skip loading the same input values for the following output channels, as shown in Fig. 3.9. When we utilize a T-word register file instead of a single register in an accumulator in a PE, this enables the PE array to reduce data accesses to at most $1/T$.

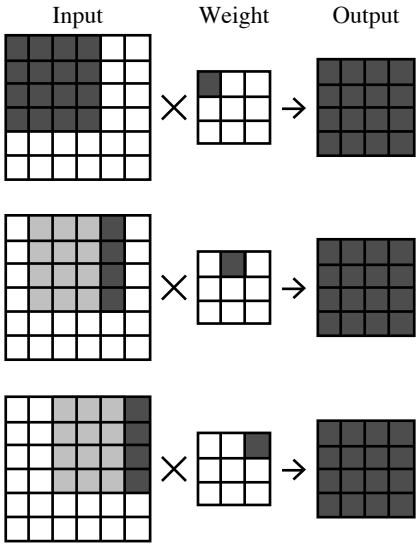

Fig. 3.8: Sequential locality in the Conv layer processing. The light-grayed elements can be reused.

3.4 Conv/FC Configurable Processor Exploiting Data Locality

We have explained the computation of Conv and FC layers on an ideal processor array with individual and shared input busses and the data reusability of CNNs on the array system. In this section, we build a feasible array architecture by exploiting the data locality that

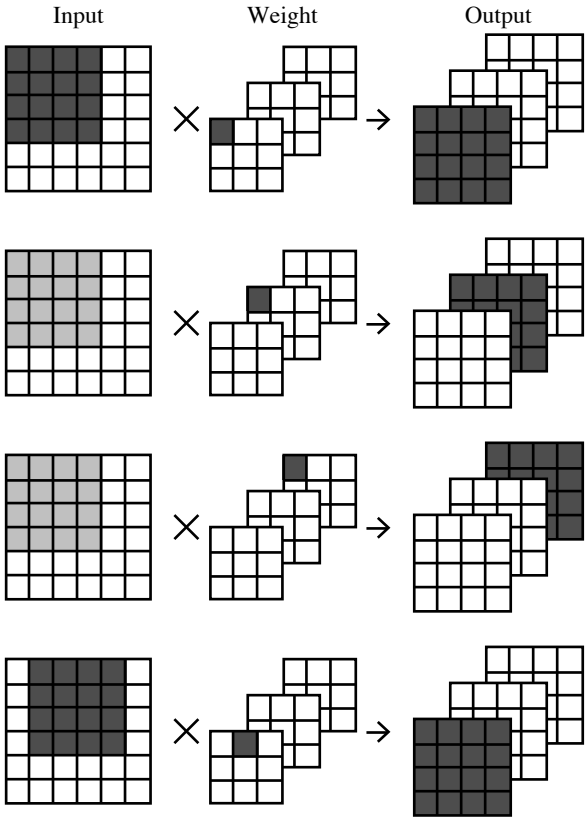

Fig. 3.9: Temporal locality in the multi-channel convolution.

enables further efficiency improvement. The multithreaded accumulator and two possible types of row-wise data delivery are proposed.

3.4.1 Shift-Based Row-Wise Data Supply

The Conv computation explained with the ideal architecture consumes square partial input area every clock cycle. However, a square area is not memory-friendly because it requires discontinuous addressing. Also, we assumed that the square partial input area is fetched within a cycle on the individual bus, which requires a very high internal bandwidth between the buffer and PE array. We introduce shift-based row-wise data delivery for simpler access patterns and a more compact architecture design as a practical implementation.

Since the memory addresses are continuous in a kernel row in Conv computation by the output-parallel PE array, it appears to appropriate that a row buffer in each PE row retains

the input rows as Fig. 3.10. With permitting the latency of filling PEs column by column, this implementation is a light-weighted one; only the PEs on the leftmost column should have the individual inputs, and the other PEs acquire the input data from their left-side PEs via the forwarding bus. Note that the overhead of filling all PEs sequentially via the forwarding bus will be discussed later in Section 3.4.2.

However, the row buffers are still not sufficient for realistic workloads. A Conv operation requires K input rows per PE row; thus, the entire PE array requires $(P_y + K - 1)$ input rows. For example, four rows indicated in gray in Fig. 3.10 are consumed by the first kernel row, then the remaining two input rows are used by the later kernel rows. Therefore, we need a mechanism for selecting the row buffer connected to the leftmost column. There are two possible solutions, namely, multi-port SRAM buffers and multi-bank ones.

Figs. 3.11 and 3.13 shows a multi-port solution using the P_y-W P_y-R multi-port SRAMs as row buffers, where a PE row can read the input from any row buffer. Compared to the ideal model (Fig. 3.4), the individual input and output buses moved from the PEs to the row buffers. This modification achieves the fewest internal memory accesses. Each input row is loaded to a row buffer only once and is read by all PE rows consumes it (at most K rows) as Fig. 3.14(a). The difficulty of this implementation is the scalability of multi-port SRAMS; the multi-port SRAMs cost $O\left(P_y^2\right)$ area.

A multi-bank solution is presented in Fig. 3.12. The PE is the same as the multi-port one in Fig. 3.13. Here, we employ the $(P + K - 1)$ row buffers; each of them is realized by a multi-bank SRAM, and the forwarding buses are looped back to connect to the row buffers of the next PE rows. Each row buffer has an input multiplexer; it acquires the input activations from the external bus in the data loading phase and the upper PE row's rightmost forwarding output in the processing phase, as described in Fig. 3.14(b). The multi-bank SRAM buffers cost only $O\left(() P_y\right)$ hardware resources and endure minimal external memory access. However, the number of internal buffers' write accesses increases compared to the multi-port solution. This is because the input row selection between the kernel rows is made by the inter-row buffer data movements instead of explicitly switching them by crossbars. The input activations that a PE row has just consumed are then written from the forwarding bus of the rightmost PE to the lower PE row's buffer.

3.4.2 Deep-Multithreaded Accumulator with an SRAM

The row-wise data delivery simplifies the internal bus structure and relaxes the external memory access; however, an overhead of filling the PE array with the input before the first kernel row computation occurs as a drawback. The frequency of data loading from the row

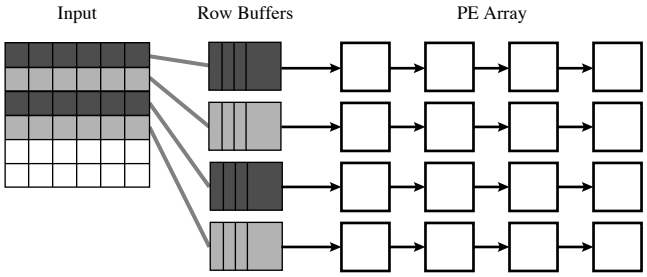

Fig. 3.10: Inpout data mapping on the shift-based array.

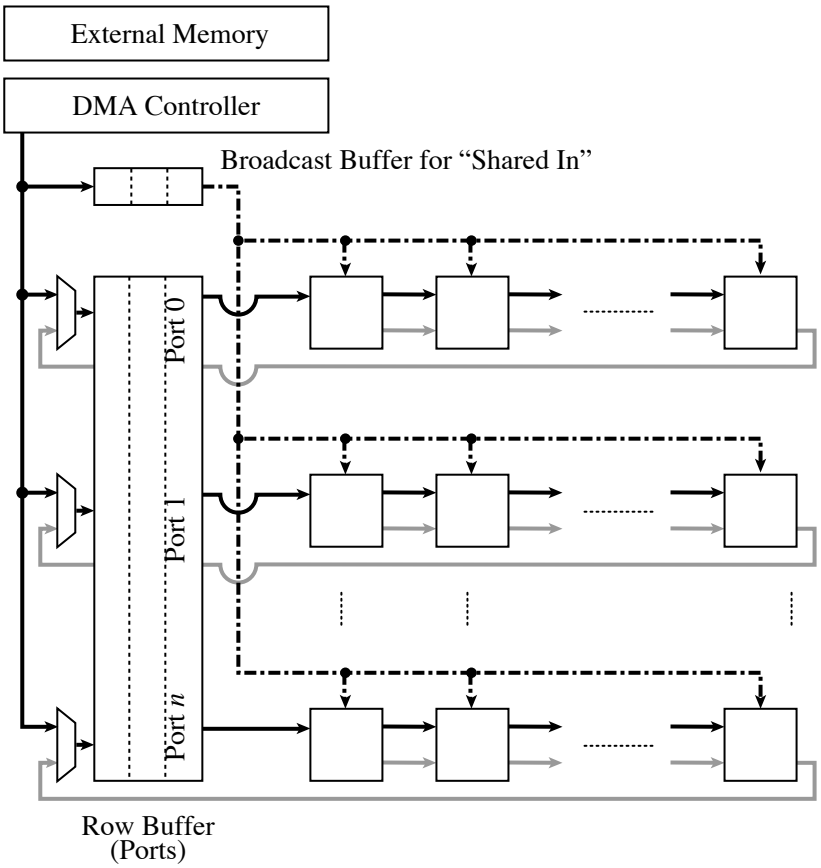

Fig. 3.11: Shift-based architecture with a multi-port row buffer.

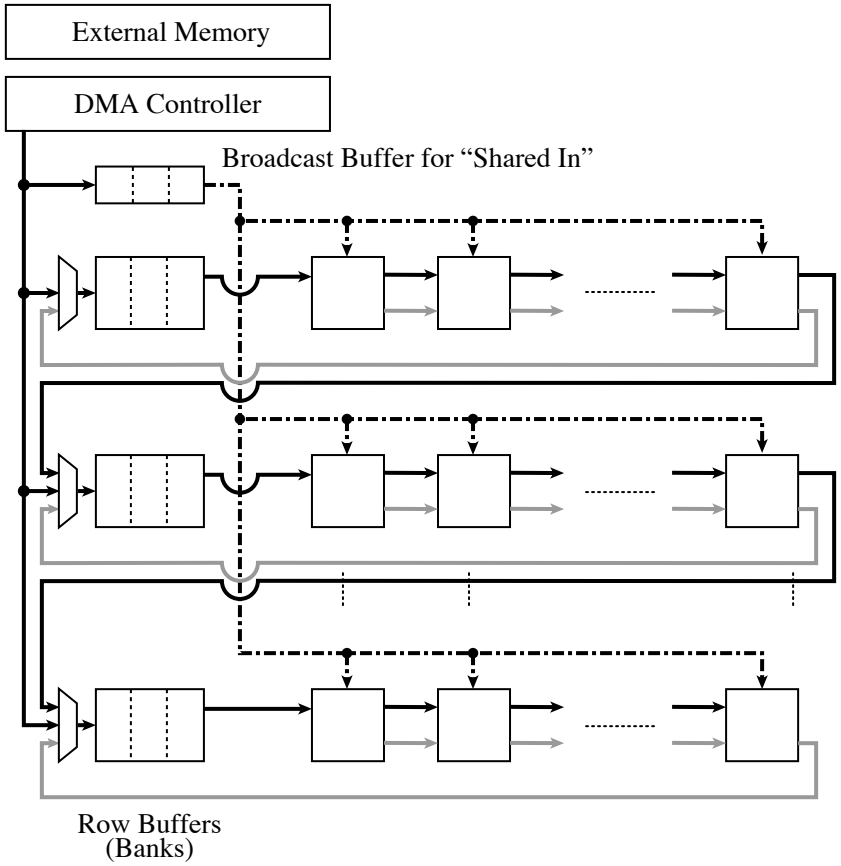

Fig. 3.12: Shift-based architecture with multi-bank row buffers.

buffer to the PE array should be minimized to overcome this overhead. Here, we attempt to extend the PE with a multithreaded accumulator to reuse the inputs the PE has loaded once as many as possible. Later layers in a convolutional neural network model have more channels and smaller plane sizes in general. Thus, sequential processing over the input channels is dominant, and the cycles for the row-wise data transfer can be amortized by it.

We consider utilizing an SRAM buffer as the accumulator's register to increase the number of channels computed using a single input channel. An SRAM is denser in terms of data capacity (*i.e.* , has higher bytes per μm^2). It is suitable for reusing an input channel for more output channels to get higher arithmetic intensity. Since the significant processing time and energy of the convolutional neural network processing is consumed in deeper layers with smaller planes and more channels, an SRAM buffer is desirable despite its

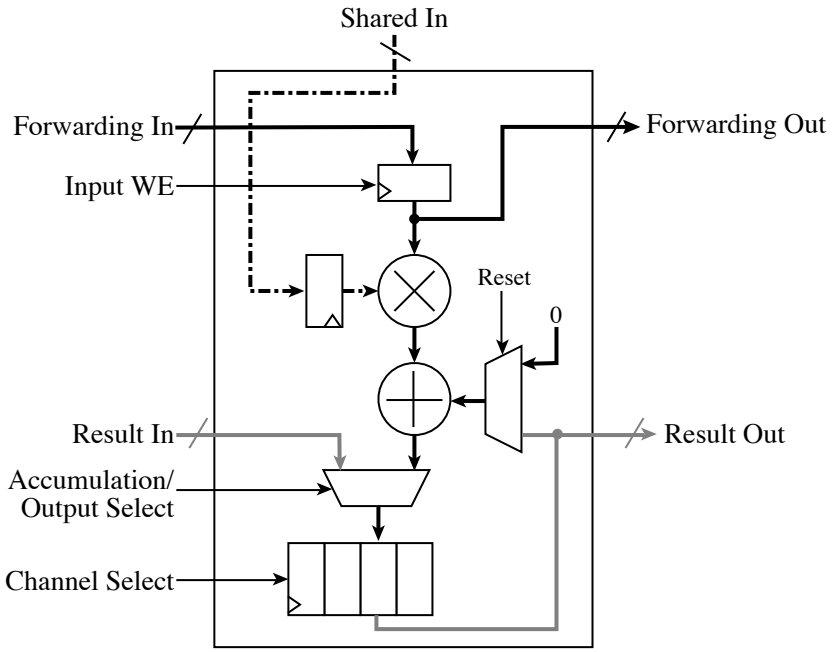

Fig. 3.13: A PE for shift-based array (Figs. 3.11 and 3.12).

larger implementation area compared to a register file assumed in the ideal architecture,

3.5 Evaluation and Discussion

3.5.1 Setup

We evaluate the Conv/FC architecture in this section. We first estimate the computation performance and arithmetic intensity, and then we discuss the complexity and required hardware resources of the row-wise data delivery and SRAM buffers.

The definition of parameters and the setup of the evaluation are the following.

- Conv layer
  - Input size: $N_{ix} \times N_{iy} \times C_i$
  - Output size: $N_{ox} \times N_{oy} \times C_o$
  - Kernel size: $K \times K \times C_i \times C_o$
- FC layer

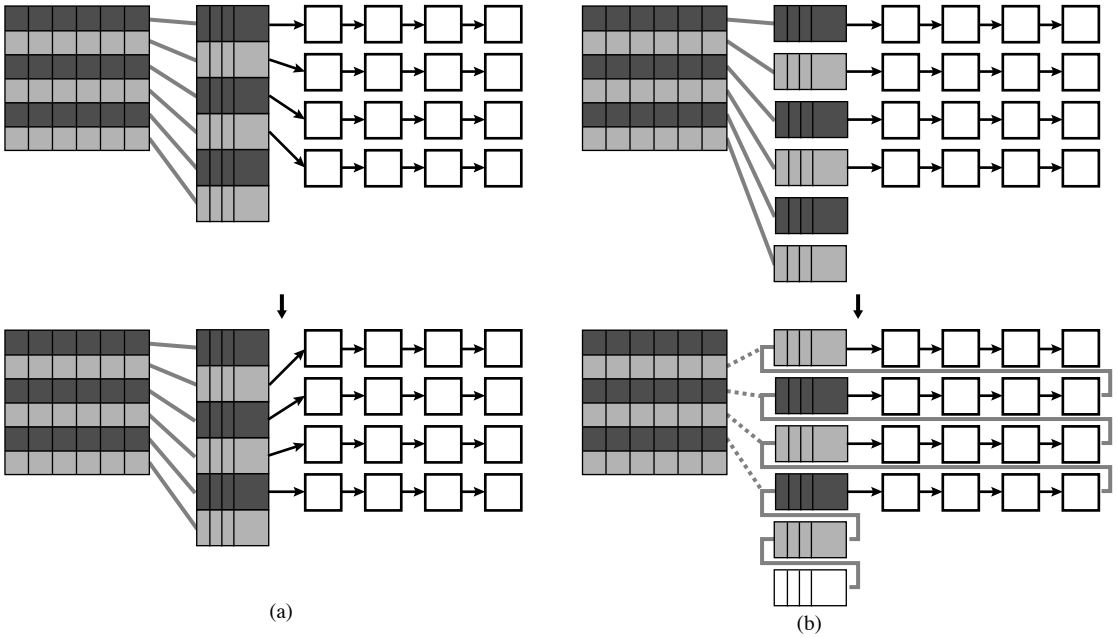

Fig. 3.14: Data mapping and row selecting on (a) the "multi-port" and (b) "multi-bank" systems.

  - Input size: N_i
  - Output size: N_o

- Hardware setup:

  - PE array size: $P_x \times P_y$
  - Number of threads in a PE accumulator: T
  - Clock frequency: f
  - Bit-width: b

3.5.2 Performance

Table 3.1 summarizes the processing time and required data rates in Conv and FC processing.

TABLE 3.1: PROCESSING TIME [CYCLES] AND DATA RATE [BPS]

	Conv	FC
Pre-requirements [a]	Out $N_{ox}N_{oy}C_o$, In $N_{ix}N_{iy}C_i$	Out N_o, In N_i
Processing Time	$(P_x + K - 1)\, KC_oC_i \left\lceil \frac{N_{ox}}{P_x} \right\rceil \left\lceil \frac{N_{oy}}{P_y} \right\rceil$	$N_iP_x \left\lceil \frac{N_o}{P_xP_y} \right\rceil$
Input Rate	$bf \frac{(P_y+K-1)N_{ix}\left\lceil \frac{C_o}{T} \right\rceil}{(P_x+K-1)KC_o\left\lceil \frac{N_{ox}}{P_x} \right\rceil}$	$bf \frac{(1+P_xP_y)}{P_x}$
Output Rate	$bf \frac{N_{ox}N_{oy}}{(P_x+K-1)KC_i\left\lceil \frac{N_{ox}}{P_x} \right\rceil \left\lceil \frac{N_{oy}}{P_y} \right\rceil}$	bfP_y/N_i

[a] Bit width is b, clock frequency is f, array size is $\left(P_x \times P_y\right)$, the number of accumulator threads is T, and kernel size is $K \times K$.

A Conv operation consumes P_x cycles per a kernel row for preloading input activation from row buffers by shifting and $(K-1)$ cycles for conducting MAC operations. The total number of this procedure is KC_iC_o for all the kernel rows and input/output channels, so the processing time of a Conv layer is:

$$(P_x + K - 1)\, KC_oC_i \quad \text{[Cycles]} \tag{3.5.1}$$

When the output size exceeds the PE array size, the processing is done in a time-divided manner, costing $\left\lceil \frac{N_{ox}}{P_x} \right\rceil \left\lceil \frac{N_{oy}}{P_y} \right\rceil$ time-division blocks. As a result, the processing time becomes:

$$(P_x + K - 1)\, KC_oC_i \left\lceil \frac{N_{ox}}{P_x} \right\rceil \left\lceil \frac{N_{oy}}{P_y} \right\rceil \quad \text{[Cycles]} \tag{3.5.2}$$

We estimate the amount of the required input data. At the beginning of each row time-division block, the input activation is transferred from the external memory to the row buffers in burst mode. The PE array consumes $(P_y + K - 1)$ rows of the input data in each of the $\left\lceil \frac{N_{oy}}{P_y} \right\rceil$ row time-division blocks, and the input data has N_{ix} columns. Unless the number of output channels C_o exceeds the number of accumulator threads T, the input data are not reloaded. Therefore, the amount of external data transfer is described as:

$$\left(P_y + K - 1\right) N_{ix} \left\lceil \frac{N_{oy}}{P_y} \right\rceil \left\lceil \frac{C_o}{T} \right\rceil \quad \text{[Data]} \tag{3.5.3}$$

The average input data rate is calculated by dividing Eq. 3.5.3 by Eq. 3.5.2 and normal-

izing the unit using the clock frequency f and bit-width b as following:

$$bf\frac{\left(P_y+K-1\right)N_{ix}\left\lceil\frac{C_o}{T}\right\rceil}{\left(P_x+K-1\right)KC_oC_i\left\lceil\frac{N_{ox}}{P_x}\right\rceil} \quad \text{[bps]} \tag{3.5.4}$$

Similarly, the output data size and data rate are calculated as:

$$N_{ox}N_{oy}C_o \quad \text{[Data]} \tag{3.5.5}$$

$$bf\frac{N_{ox}N_{oy}}{\left(P_x+K-1\right)KC_i\left\lceil\frac{N_{ox}}{P_x}\right\rceil\left\lceil\frac{N_{oy}}{P_y}\right\rceil} \quad \text{[bps]} \tag{3.5.6}$$

We then evaluate the performance and data rates of the FC processing. In the FC processing, since the output neurons are mapped on the PE array without respect to its 2-dimensional structure, the number of time-division blocks is $\left\lceil\frac{N_o}{P_xP_y}\right\rceil$. Each of the P_xP_y output neurons being computed on the PE array requires a weight value per cycle, corresponding to the input activation fed via the shared bus cycle by cycle. It costs P_x cycles to load a set of weight values via the row-wise forwarding bus, and this process should be repeated N_i times for acquiring all the input activation values with different weight values. The processing time of an FC layer is denoted as:

$$N_iP_x\left\lceil\frac{N_o}{P_xP_y}\right\rceil \tag{3.5.7}$$

One iteration of this process consumes P_xP_y weight values and an input activation value. As mentioned above, the number of iteration is N_i in each of the $\left\lceil\frac{N_o}{P_xP_y}\right\rceil$ time-division blocks. Thus, the total amount of loaded data throughout the FC layer processing is:

$$N_i(P_xP_y+1)\left\lceil\frac{N_o}{P_xP_y}\right\rceil \quad \text{[Data]} \tag{3.5.8}$$

The P output data are produced every time-division block, and the total output data amount is:

$$(P_xP_y+1)\left\lceil\frac{N_o}{P_xP_y}\right\rceil \quad \text{[Data]} \tag{3.5.9}$$

Therefore, the required input data rate is calculated as:

$$bf\frac{P_xP_y+1}{P_x} \quad \text{[bps]} \tag{3.5.10}$$

and the output data rate is:

$$bf\frac{P_y}{N_i} \quad \text{[bps]} \tag{3.5.11}$$

We evaluated the theoretical processing time and data rates of the proposed architecture with AlexNet, as shown in Table 3.2. The PE array size is assumed $P_x \times P_y = 16 \times 16$, the bit-width is $b = 16$, and the clock frequency is $f = 200$ MHz. This system can process AlexNet in 380 msec (2.6 fps), and the maximum required data rate is 16 MB/s, which is an acceptable result for typical embedded use cases.

Table 3.2: Evaluation of the proposed architecture using AlexNet[43]

#	Type	Weight Shape	Output Shape	Cycles	Rate [GB/s]	Util. [%]
1	Conv	$(11^2 \times 3 \times 48) \times 2$	$(224 \times 224 \times 48) \times 2$	16,144,128	0.013	100.0
2	Conv	$(5^2 \times 48 \times 128) \times 2$	$(27 \times 27 \times 128) \times 2$	19,660,800	0.009	73.9
3	Conv	$3^2 \times (128 \times 2) \times (192 \times 2)$	$(13 \times 13 \times 192) \times 2$	21,233,664	0.005	71.2
4	Conv	$(3^2 \times 192 \times 192) \times 2$	$(13 \times 13 \times 192) \times 2$	3,981,312	0.010	66.0
5	Conv	$(3^2 \times 192 \times 128) \times 2$	$(13 \times 13 \times 128) \times 2$	2,654,208	0.016	66.0
6	FC	$43,264 \times 4,096$	$4,096$	11,075,584	6.425	100.0
7	FC	$4,096 \times 4,096$	$4,096$	1,048,576	6.425	100.0
8	FC	$4,096 \times 1,000$	$1,000$	262,144	6.425	97.7
Total				76,060,416		

The PE utilization means the average ratio of working PEs to all the PEs throughout the layer processing, which is estimated by the time integration of the product of the spatial and temporal ratios. If the output size is indivisible by the PE array size, the last time-division block has inactive PEs, as shown in Fig. 3.15. Here, we define the numbers of active PE columns/rows in the last row/column time-division blocks as R_x/R_y, respectively, as follows:

$$R_x = N_{ox} - P_x \left\lfloor \frac{N_{ox}}{P_x} \right\rfloor \tag{3.5.12}$$

$$R_y = N_{oy} - P_y \left\lfloor \frac{N_{oy}}{P_y} \right\rfloor \tag{3.5.13}$$

We can then calculate the effective average PE utilization in a Conv layer as the following

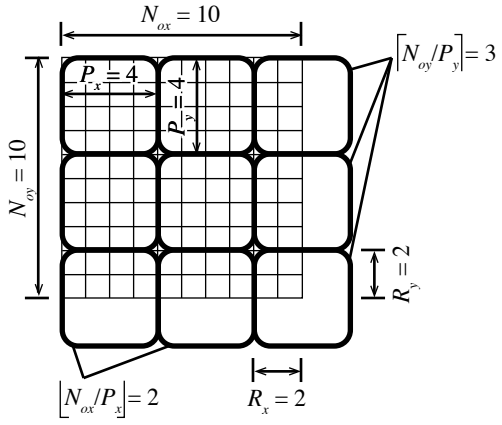

Fig. 3.15: Example of time division of the output plane. In the case of $N_{ox} = N_{oy} = 10$ and $P_x = P_y = 4$, the number of remainder pixels is $R_x = R_y = 2$, the number of fully utilized blocks is $\lfloor N_{ox}/P_x \rfloor \lfloor N_{oy}/P_y \rfloor$, and the total number of blocks is $\lceil N_{ox}/P_x \rceil \lceil N_{oy}/P_y \rceil$.

according to Fig. 3.15:

$$
\begin{aligned}
\text{(PE utilization)} = & \int \text{(spatial ratio)} \times \text{(temporal ratio)} dt \\
= & \frac{P_x P_y}{P_x P_y} \frac{\left\lfloor \frac{N_{ox}}{P_x} \right\rfloor \left\lfloor \frac{N_{oy}}{P_y} \right\rfloor}{\left\lceil \frac{N_{ox}}{P_x} \right\rceil \left\lceil \frac{N_{oy}}{P_y} \right\rceil} \\
& + \frac{R_x P_y}{P_x P_y} \frac{\left(\left\lceil \frac{N_{ox}}{P_x} \right\rceil - \left\lfloor \frac{N_{ox}}{P_x} \right\rfloor \right) \left\lfloor \frac{N_{oy}}{P_y} \right\rfloor}{\left\lceil \frac{N_{ox}}{P_x} \right\rceil \left\lceil \frac{N_{oy}}{P_y} \right\rceil} \\
& + \frac{P_x R_y}{P_x P_y} \frac{\left\lfloor \frac{N_{ox}}{P_x} \right\rfloor \left(\left\lceil \frac{N_{oy}}{P_y} \right\rceil - \left\lfloor \frac{N_{oy}}{P_y} \right\rfloor \right)}{\left\lceil \frac{N_{ox}}{P_x} \right\rceil \left\lceil \frac{N_{oy}}{P_y} \right\rceil} \\
& + \frac{R_x R_y}{P_x P_y} \frac{\left(\left\lceil \frac{N_{ox}}{P_x} \right\rceil - \left\lfloor \frac{N_{oy}}{P_y} \right\rfloor \right) \left(\left\lceil \frac{N_{oy}}{P_y} \right\rceil - \left\lfloor \frac{N_{oy}}{P_y} \right\rfloor \right)}{\left\lceil \frac{N_{ox}}{P_x} \right\rceil \left\lceil \frac{N_{oy}}{P_y} \right\rceil}
\end{aligned}
\tag{3.5.14}
$$

Figs. 3.16 and 3.17 show the relationship between the output size and processing time/resource with the PE array size $P_x \times P_y = 16 \times 16$. The processing time is proportional to the number of output channels and the output plane size, while the required data rate is almost independent of the problem size. When the problem becomes more

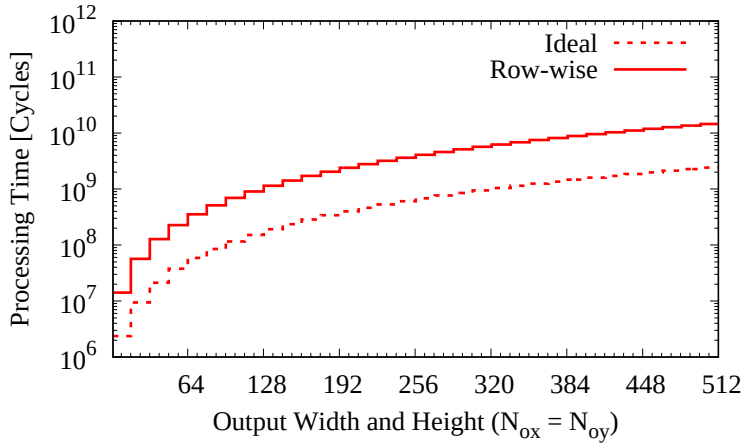

Fig. 3.16: Processing time and output data rate over the number of output channels. ($C_i = 256, N_{ox} = N_{oy} = 128$)

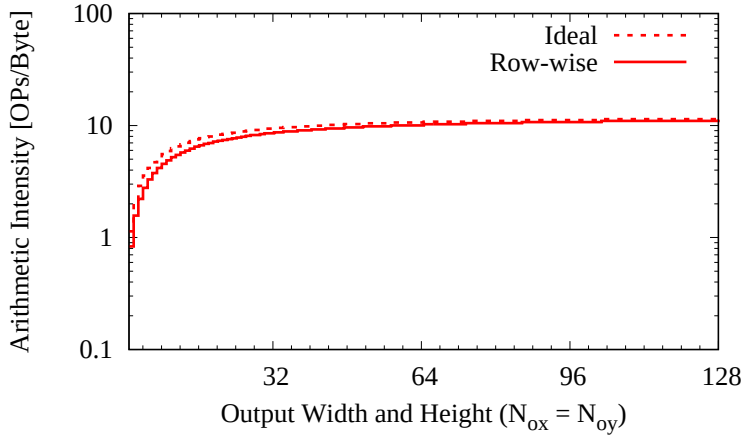

Fig. 3.17: Processing time and output data rate over the output plane size. ($C_i = 256, C_o = 384$)

extensive, the processing time increases, but the required resources such as data rate are unaffected; therefore, the availability is retained over the problem size.

Fig. 3.18 evaluates the processing times of the ideal (Section 3.3) and row-wise (Sec-

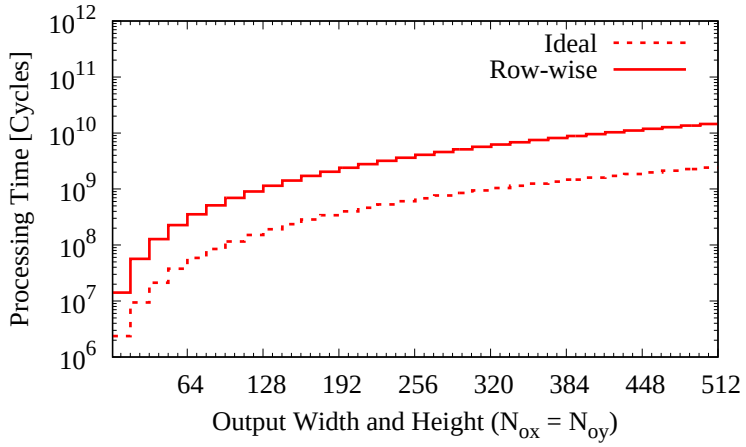

Fig. 3.18: Processing time of the ideal and row-wise architectures over the output size. $K = 3, C_i = 512, C_o = 512, P_x = P_y = 16, T = 512$.

tion 3.4) architectures when the output size of a Conv layer scales. There are no differences in the processing time and data rates between the multi-port and multi-bank row-wise architectures, as their only difference lies in the inter-row internal data selection. Compared to the ideal model, the row-wise model requires a longer processing time because the row-wise data loading is needed prior to the computation. The arithmetic intensity, indicated in Fig. 3.19, is identical among the ideal and row-wise implementations since the number of operations and the data transferring amount do not differ with the same problem to solve. In the row-wise architecture, the input data rate is relaxed, as shown in Fig. 3.20. Since both the processing time and the required data amount is linearly proportional to the output data size $N_{ox}N_{oy}$, the data rate converges to a constant.

3.5.3 Arithmetic Intensity

Let us discuss the effect on the arithmetic of adopting the multithreaded accumulator. Fig. 3.21 plots the arithmetic intensity over the output size (square: $N_{ox}N_{oy}$) in several configurations of the numbers of channels C_i, C_o and threads T. The fine three lines with different numbers of accumulator threads assume fewer input/output channels, typical configurations in shallower layers in convolutional neural networks. These shallower layers with fewer channels usually have larger output sizes, which are saturated in the figure; the arithmetic intensity improves orders of magnitude when the number of threads

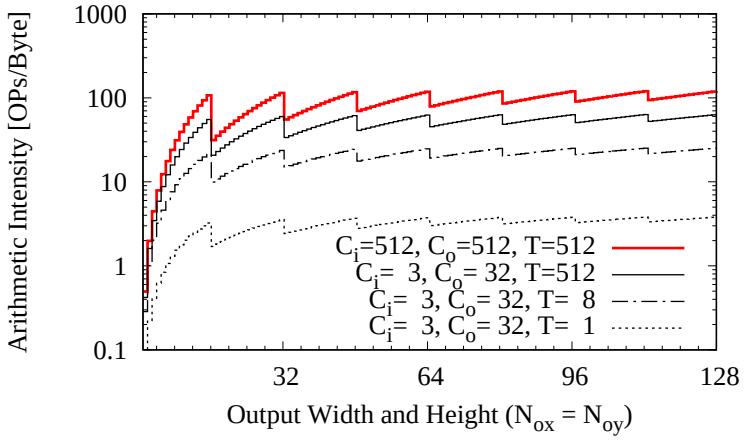

Fig. 3.19: Arithmetic intensity of the ideal and row-wise architectures over the output size. $K = 3, C_i = 512, C_o = 512, P_x = P_y = 16, T = 512$.

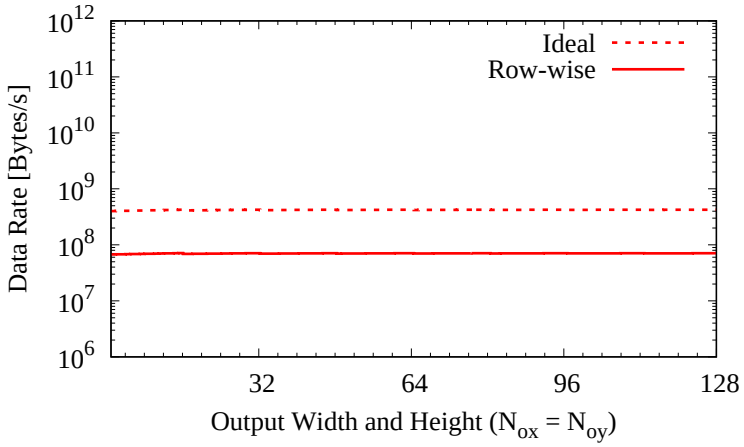

Fig. 3.20: Required input data rates of the ideal and row-wise architectures over the output size. $K = 3, C_i = 512, C_o = 512, P_x = P_y = 16, T = 512$.

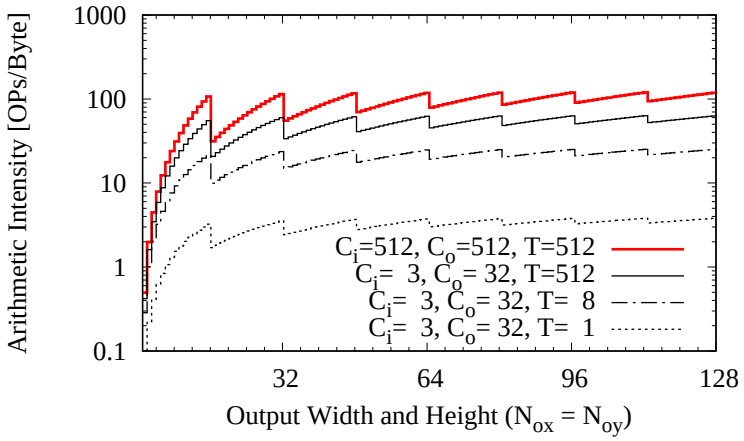

Fig. 3.21: Arithmetic intensity over the output size with several configurations of the numbers of channels and threads. $P_x = P_y = 16, K = 3$

T increases from 1 and 8 to 512. On the other hand, the number of channels grows larger in the deeper layers, and the output plane size gets small, like the left part of the bold line in Fig. 3.21. In such a situation, as the opportunity of reusing the kernel weight values decreases, the arithmetic intensity degrades. However, the multithreaded accumulator works effectively even in those configurations.

3.5.4 Area of the Multi-Port and Multi-Bank Implementations

We introduced two possible structures of the row buffers, the multi-port and multi-bank ones, in Section 3.4, and evaluated the performance of the architecture in Section 3.5.2. In this subsection, we discuss the complexity of these implementations.

As described in Section 3.4, a PE row acquires K input rows; as a result, the entire PE array consumes $(P_y + K - 1)$ input rows. These $(P_y + K - 1)$ input rows are stored in the P row buffers, and there is a need for a mechanism to select a row to feed into each PE row. We introduced two SRAM types as the row buffer with data selection, the multi-port and multi-bank SRAMs. The multi-port one can select any input row for each PE row, while the multi-bank one do the data selection by moving the stored data with the connections fixed.

If we utilize $(P_y + K - 1)$-R SRAMs as the multi-port row buffers, the input activation data are loaded to the SRAM only once, and they are read by at most K rows of PEs (*i.e.* ,

TABLE 3.3: ENERGY COMPARISON OF MULTI-PORT AND MULTI-BANK IMPLEMENTATIONS

P_y	Multi-port			Multi-bank		
	# Ports	Area [mm^2]	Energy [nJ]	# Banks	Area [mm^2]	Energy [nJ]
2	4	0.694	0.706	4	0.085	0.103
4	8	4.726	5.041	6	0.127	0.279
6	8	4.726	9.410	8	0.169	0.535
14	16	41.332	105.697	16	0.338	2.387

K output rows). The write accesses of the SRAMs are required only once at the beginning of the time-division block. However, an n-R SRAM requires $O\left(n^2\right)$ of implementation area in general. The required capacity and number of ports of the row-wise SRAM buffer are in order of $O\left(P_y\right)$. Since the implementation area of an SRAM is in proportion to capacity and the square of the number of ports, the total SRAM implementation area of the PE array becomes $O\left(P_y^3\right)$.

On the other hand, if we utilize $(P + K)$ sets of 1-R 1-W SRAMs as the multi-bank solution, the SRAM area footprint is $O\left(P_y\right)$ with the P_y PE rows. The increase of SRAM banks is basically the increase of SRAM macros; its area is proportional to the number of banks. In this solution, the leftmost PE column pops a word from the row buffer, and the rightmost PE column's forwarding output writes a word to the next rows' buffers in each cycle. Thus, although the data load from the external memory to the row buffers is needed only once at the beginning of the time-division row, just like the multi-port solution, the internal SRAM write accesses increase by K times.

Table 3.3 shows the area and energy specification of the 16-bit 512-word multi-port and multi-bank SRAMs simulated using CACTI [1]. We estimated the energy of memory accesses to process a Conv layer with C=1, assuming two additional banks or ports to the number of PE rows P_y are needed for the overlapped external memory access during the multithreaded accumulation. The internal data movement needed in the multi-bank solution is included. The power consumption of the read and write accesses are supposed to be equal, according to [62]. The simulation shows that, despite the internal SRAM accesses increase, the multi-bank solution consumes less energy than the multi-port implementation. The number of SRAM ports/banks in the practical architecture should be carefully determined considering the PE array size (P_x, P_y), the throughput and latency of the external memory, and the target application (K, C_o, C_i, *etc.*).

3.5.5 Discussion on Memory Usage

In Conv processing, the PE array requires P_y input data at the first output channel in the multithreaded accumulators; therefore, the input data rate requirement of the PE array (internal memory bandwidth of the row buffers) is P_y/T [Words/Cycle]. It should be noted that, when the output channel C_o is indivisible by the number of threads T and leaves C_{ro} ($= C_o \mod T$) residual channels, the read interval is shortened to $C_{ro} < T$, resulting in tightening the internal bandwidth requirement to P_y/C_o [Words/Cycle]. It is possibly solved by forcing the multithreaded MAC to always take T cycles even if C_o cannot be divided by T. If the read interval T or C_{ro} is greater than P_xP_y, the number of input data to be loaded to the PE array, then serial data load can be utilized with the latency of data loading amortized, instead of the row-wise data distribution. The P_xP_yT output values of the PE array become valid when all the input channels and kernel elements are MACed, *i.e.*, every K^2C_iT cycles; the required internal output data rate is $\frac{P_xP_y}{K^2C_i}$ [Words/Cycle]. A weight value is loaded cycle by cycle during the MAC operation; the internal data rate of the shared bus is always 1 [Words/Cycle]. These observations indicate that the internal input data rate becomes severe when the number of the output channel C_{ro} is small and that the output data rate is tighter when the number of input channels C_i or kernel size K is smaller.

We discuss the optimal configuration of the SRAMs for the row buffers. For the latency of loading input data from the external memory to be amortized, the processing time must exceed the memory access time. We need to allow the external accesses overlapped to the MAC operations to fulfill this condition. When the number of threads T is large enough, the external data load can be completed during the multithreaded MAC operation, in which the read accesses from the row buffers do not occur except for the first cycle of each block. Therefore, the SRAM depth should be at least $2P_x$ to store the currently used partial input area while the next area is transferred in the background. For further data reuse, the row buffers with larger depth are preferred to retain the solid area of the input activation, *i.e.* , the entire time-division row block, not only a block.

The implementation area of the PE array is proportional to the number of threads T, and the external data rate and the area of SRAM buffers are proportional to the number of PE rows P_y and in inverse proportion to T. On the other hand, the processing time is almost proportional to the number of threads T and the inverse of the PE array size $1/P_xP_y$. On the construction of a practical system, we should consider these characteristics according to the requirements of the application.

3.5.6 Stationary

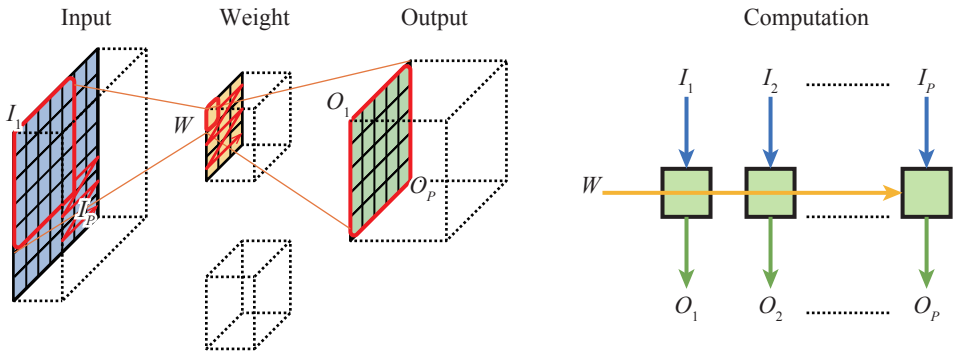

Fig. 3.22: SOC-MOP output-stationary computation [17, 12].

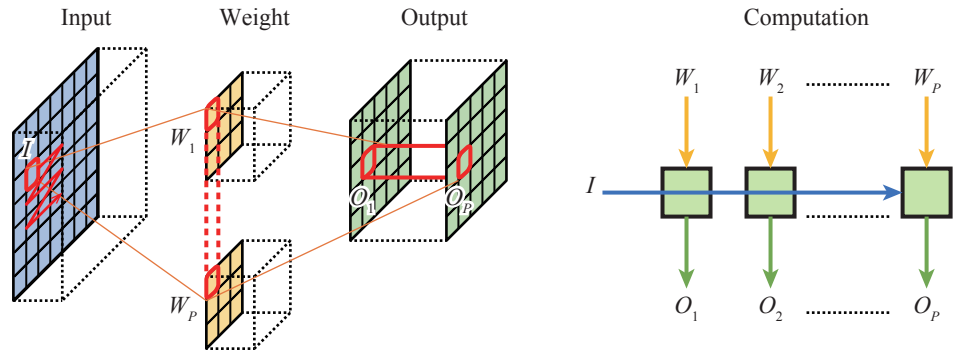

Fig. 3.23: MOC-SOP output-stationary computation [17, 12].

Parallel processing architectures for neural networks can be categorized into several groups by their "stationary" [17, 12]. Generally, a PE in a parallel architecture acquires input activation and weight values to produce the output activation; some are mapped spatially on multiple PEs while the others flow on the PEs temporally. The term "stationary" is a measure representing what is mapped spatially to stay on PEs and be reused

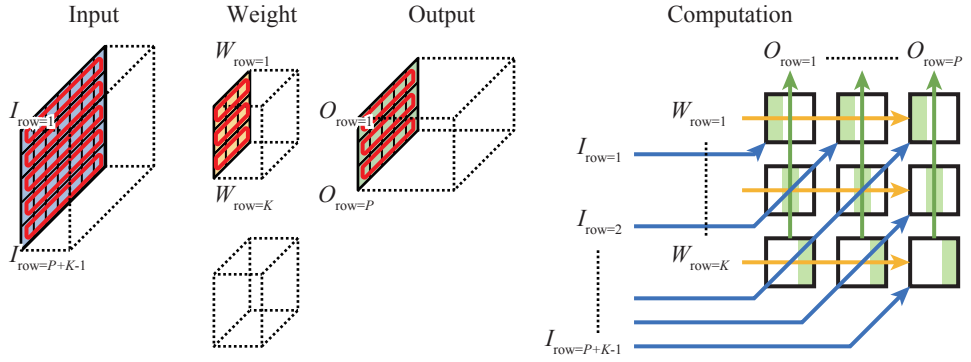

Fig. 3.24: Row-stationary computation [17, 12].

for multiple cycles. Eyeriss [17, 12] characterized itself as a *row-stationary* architecture, where a "row" of the output activations remains on a PE throughout a transaction. The architecture we have discussed in this chapter is an *output-stationary* one, where a PE is bound to an output pixel (or neuron) and consumes the inputs and weights in multiple cycles.

They divided output-stationary architectures into several subgroups by the axes of output activations to be computed in parallel, that is, *SOC-MOP* (single-output-channel multiple-output-pixel), *MOC-MOP* (multiple-output-channel multiple-output-pixel), and *MOC-SOP* (multiple-output-channel single-output-pixel). The architecture constructed in this chapter can be interpreted as an output-stationary architecture switching the SOC-MOP and MOC-SOP modes appropriately for Conv and FC layers.

SOC-MOP output-stationary, shown in Fig. 3.22, represents an architecture where the multiple PEs process multiple output pixels in a single channel. It can exploit the weight reusability because a weight value is shared among all the pixels (*i.e.*, among the entire PE array) in each channel. This is suitable for Conv layer processing, where weight sharing is a fundamental characteristic. Our architecture computes Conv layers in this mode by distributing the input values to the PEs via the row buffer while the weight values are scanned sequentially. Although Eyeriss and our architecture do not utilize the sparsity, this configuration is also efficient for sparse neural networks, where the location of valid values in a weight matrix is irregularly distributed.

On the other hand, MOC-SOP output-stationary mode (Fig. 3.23), where a PE corresponds to a pixel in each channel, can utilize the input reusability because an input channel is used to compute all the output channels. By replacing the word "channel" in this expla-

nation with "neuron," this stationary model can efficiently house an FC layer. As an FC layer does not have weight reusability in a single frame, this is the only way to exploit the data reusability in an FC layer. Encouraged by this observation, our architecture utilizes this stationary mode for FC layers with the weight values fed via the row buffers in parallel and the input value provided sequentially. This mode is also efficient for randomly pruned input activation or point-wise convolution that were proposed later, though they are not considered in our architecture and Eyeriss.

Eyeriss advocated the row-stationary architecture (Fig. 3.24) to exploit both the input and weight reusability. A row in a kernel is shared among the PEs horizontally (corresponding to multiple output rows), and a row of the input activation is shared diagonally. As a result, each of the PEs aligned vertically obtains the partial sum for an output row calculated from an input row. The final sum, *i.e.*, the sums for an output row from all the input rows in a kernel, is calculated by collecting and summing up the partial sums vertically. However, as an FC layer does not have the weight reusability in a single transaction, the row-stationary architecture cannot use the horizontal weight reusing. Here, it is almost equivalent to the MOC-SOP output-stationary type; as a possible difference from the output-stationary one, the partial sums from different subsets of input neurons could be computed on vertically-aligned multiple PEs in a distributed manner.

Hence, Eyeriss is an architecture putting great importance on Conv layer processing with exploiting the input and weight reusability as much as possible. In contrast, the discussed architecture houses both Conv and FC layers by switching the SOC-MOP and MOC-SOP output-stationary modes. Moreover, the architecture further tries to exploit the input reusability in Conv mode in an "input-stationary" flavor by retaining the input values for multiple output channels temporally with the multithreaded accumulators.

3.6 Conclusion

This chapter constructed a CGRA architecture for efficient and flexible processing of Conv and FC layers. We extracted the data pattern of one-to-all parallel MAC as an essential primitive laying in both Conv and FC layers. We built a parallel MAC array employing shared and individual buses to exploit this data pattern, which can process Conv and FC layers in a single structure by changing only the data delivery. To realize a feasible memory architecture for this data delivery, we adopted multi-bank row buffers and multithreaded accumulators, which relaxes the hardware complexity and memory bandwidth requirements. We also conducted an architecture exploration assuming several realistic workloads in convolutional neural networks starting from the prototyped architecture.

This chapter's main contributions include the flexibly programmable architecture exploiting the algorithms' underlying data flow characteristics, the memory-data-pattern-centered reconfigurable architecture, the architecture optimization using hierarchical memory, along with comprehensive workload analysis.

Chapter 4

BRein Memory: Near-Memory Processor for Quantized Neural Networks

4.1 Introduction

In Chapter 3, we looked into the dataflow architecture of a typical convolutional neural network processor. The evaluation results showed that the efficiency is more affected by the amount and patterns of data transfer than computational resources, especially for dense fully-connected layers. There are also reports that the energy consumed for data transfer is much larger than the computation [12].

These facts suggest that if we can integrate the computational units nearby the internal memory to eliminate the external memory, the energy efficiency and performance could be improved. This chapter aims to improve the efficiency of dense neural network models constructing a fully-pipelined near-memory processor architecture. Since all the parallel computation and data movements are enclosed near memory, no expensive data transfer is required. The computational units can be pipelined without scratchpad memory by matching the computation procedure and memory addressing a neural network layer's parallelism.

The computational units should be compact enough to be integrated nearby memory macros. Low-precision quantization techniques have been proposed and widely used in commercial/prototype architectures to realize simple hardware implementation; the activation and weight data are expressed, stored, and computed in a fewer bit width. Fixed-point, logarithmic, and even binary quantization are researched for edge-side processing, as simpler circuits can realize them than the floating-point expression commonly used

in CPU/GPU processing [83]. In 2015, a neural network model that uses binary weight and full-range activation was proposed [20]. The binarized weights take only +1 or −1, which reduces the memory footprint and relaxes the computational complexity. Since the multiplication between a binary weight and a full-range activation produces only an identical or sign inversion of the activation, multiply-accumulation (MAC) is replaced by simple addition. A neural network algorithm binarizes both the activation and weight, named "binary neural network," was presented in 2016 [18]. All of the activation values are also binarized into ±1, in addition to the weights. Here, multiplication between input and weight is replaced with a simple XNOR operation. We modify this algorithm to use in our near-memory neural network processor.

This chapter is based on the previous conference and journal papers [3, 5, 8]. Section 4.2 lists prior work, including reconfigurable or dadicated neural network/neuromorphic accelerators. In Section 4.3, we explain our in-memory accelerator architecture's base idea by reviewing the basics of conventional and binary neural networks. Section 4.4 shows the entire architecture structure, introduces the reconfigurable extensions for versatile DNN accelerations, and details our highly parallel computation mechanisms. In Section 4.5, we show the effectiveness of our proposed system by evaluating the prototyped chip and comparing it to CPU, GPU, FPGA, and prior hardware solutions. Section 4.6 shows the application examinations conducted on the prototyped chip, including the setup of training/preprocessing software environments. Here, we also consider the requirements for extending the architecture based on the observations of the current prototyped architecture. Finally, we conclude the chapter in Section 4.7.

4.2 Related Work

Many studies have been conducted on the neural network inference accelerator built on FPGAs or application-specified integrated circuits (ASICs). The Eyeriss architecture [17] proposed by MIT in 2016 is one of the most successful processors for convolutional neural networks (CNNs), and features a single-instruction multiple-data (SIMD)-style array with multilevel on-chip memories and efficient data delivery by a network-on-chip (NoC).

Envision [55] by KU Leuven, and their previous work [57] are dedicated to CNN processing utilizing a SIMD multiply-accumulator (MAC) array with technical task division and a reconfigurable voltage control to reduce power consumption. In 2016, KAIST proposed a SIMD MAC-based CNN accelerator [75], which utilizes the dynamic precision control, and some techniques to reduce the kernel data amount. Its improved version DNPU [73] is one of the few accelerators with the ability to process recurrent neural

networks (RNNs) as well as CNNs.

TrueNorth [52, 51], developed by IBM, is a neuromorphic accelerator, which employs in-memory spiking neuron cores to simulate the asynchronous neural behavior. It works at low power based on the sparsity of the asynchronous synaptic activities in "real time" (the clock frequency of the real brain).

From the viewpoint of the algorithm, our approach originates from the computing of low-precision neural networks. The concept of processing DNNs at low computational precision has been researched over several years [19], exploiting the redundancy in their large amount of weights. The first report on neural networks using binary weights was "BinaryConnect" [20], which restricts all the weight to ±1 while the activations are still in full-precision, proving that this extreme approximation could obtain a high recognition accuracy, thus encouraging the later hardware-friendly low-precision algorithms. The technique binarizing both weights and activations [22], termed as "BinaryNet" [18], was reported in 2016, followed by "XNOR-Net" [68] that expresses the weights with binary weights and a few representative real values. In the second half of 2016, "ternary weight network" [49], which restricts the weights to be −1, +1, or 0, was presented and proved to achieve an even smaller accuracy degradation compared to binarization in some DNN categories because of its richer numerical representation [49, 35]. The growth of these low-precision neural networks cannot be stated without stating the contribution of the "Batch Normalization" algorithm [37], which tolerates the influence of the changes in the input distribution by applying a statistical transformation in the training phase.

4.3 Binary DNN Algorithm and Base Architecture

4.3.1 Motivation

A modern DNN model is usually composed of dozens of millions of parameters (synapse weights), which are multiplied with the input activations and then accumulated. A large number of MAC operations are needed to generate the output from these input activations. Neural network processing requires high memory capacity and high computational performance.

Many ASIC- and FPGA-based neural network accelerators have been proposed to speed up neural network computation and achieve high energy efficiency. The major structure of such accelerators is an "application-specific processor," as shown in Fig. 4.1; this is made up of dedicated processing units and an external memory to store the parameters of a neural network. However, the demerit of such a type of structure is that it often suffers

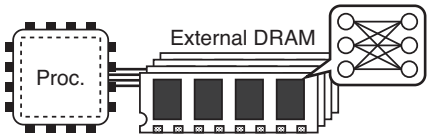

Fig. 4.1: Conventional DNN accelerator: optimized processor with high-capacity external DRAM.

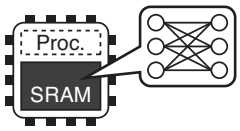

Fig. 4.2: Concept: in-memory single chip solution.

from the bottleneck of the external memory (or "Von Neumann bottleneck"), where the throughput and energy efficiency are limited by the external DRAM accesses.

To solve this problem, we considered a single-chip in-memory approach, as shown in Fig. 4.2, in which all the network parameters are stored in on-chip SRAMs and the computation completes in logic units located near the SRAMs. This "in-memory" concept excludes costly (in time and energy) external DRAM accesses and could improve the efficiency of neural network processing. The key concept of this approach is the integration of the processing circuits that harmonize with the nature of the SRAMs, as well as the method of sufficiently compressing the neural network parameters to store in on-chip SRAMs. We must exploit the essential intra-word (bit-wise) parallelism of SRAMs and hide the sequential behavior of inter-word (word-wise) accesses to achieve highly efficient in-memory computation (Fig. 4.3).

It should be noted, in addition, that the architectural concept itself does not limit the on-chip memories to SRAMs; it could be utilized with any type of memories such as embedded DRAMs or embedded non-volatile memories, if the area, process technology, power, and the timing specification were sufficient to its application. It would also be possible to incorporate the proposed architecture into DRAM or non-volatile memory chips for realizing binary/ternary DNN acceleration inside a memory system.

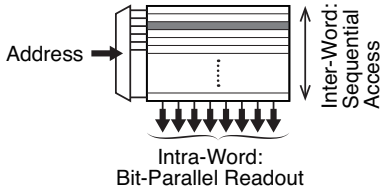

Fig. 4.3: Parallelism in an SRAM. SRAM has the intra-word bit-parallel readout and inter-word wise serial access.

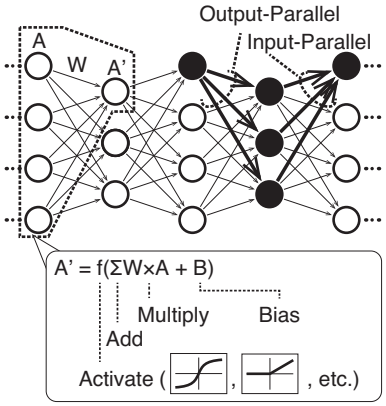

Fig. 4.4: Computation of a DNN.

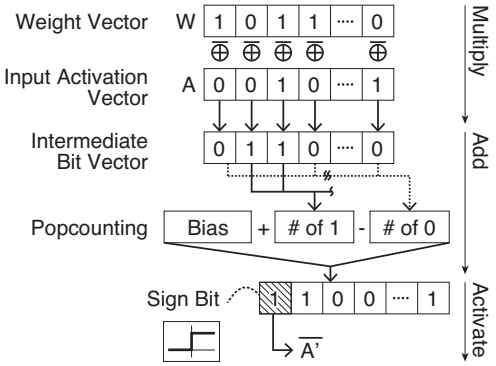

Fig. 4.5: Neural function in a binary DNN.

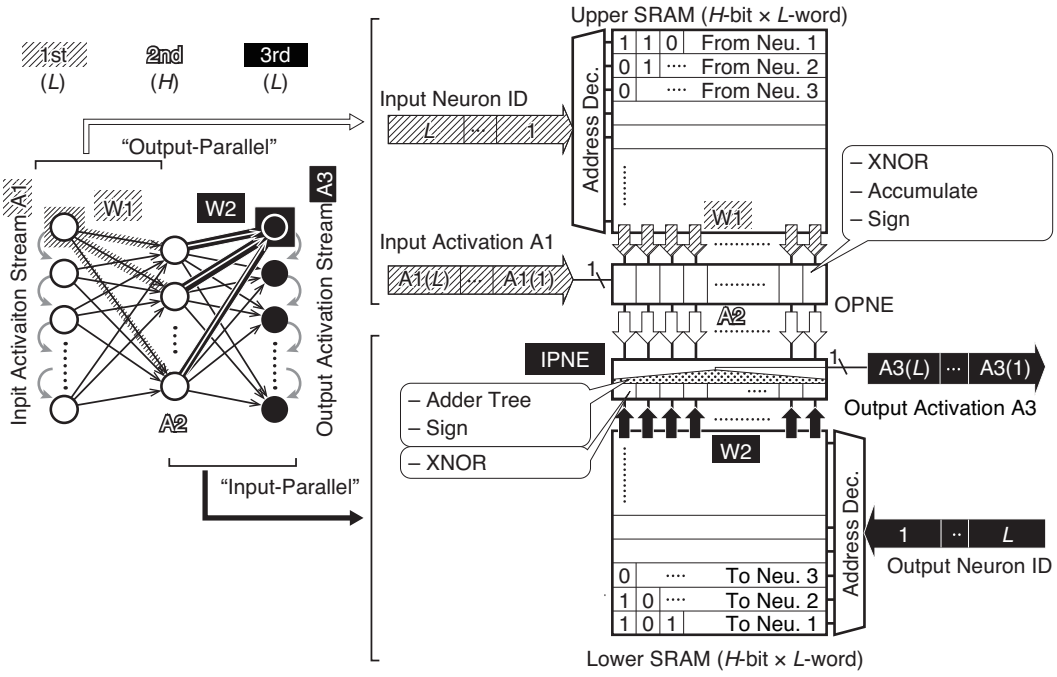

Fig. 4.6: Processing-in-memory module (PIM) with output- and input-parallel processing engines (OPNE and IPNE) for the 3-layer BNN processing.

4.3.2 Binary Neural Networks and Its Computation

Fig. 4.4 indicates a general DNN process. A "layer" consists of neurons with inputs from the previous layer and outputs to the next layer. The activation of a neuron is calculated by 1) multiplying the input activations with their corresponding weights, 2) summing up the products, and then 3) applying a nonlinear function ("activation function").

Binary neural networks [68] restrict ("binarize") all the activations and weights to be either $+1$ or -1. When -1 is denoted as 0, a binarized neural function becomes 1) performing bit-wise XNOR of the weight and input activation vectors ("multipling"), 2) counting the numbers of 1s and 0s in the XNORed intermediate bit vector ("adding"), and then 3) determining the output activation as 1 (0) when the 1s (0s) are dominant ("activation function"), as shown in Fig. 4.5.

There are two opportunities of parallel computation in a DNN: output-parallel (a single activation to multiple inputs) and input-parallel (multiple activations to a single input), as shown in Fig. 4.4. This indicates that if the output-parallel computation (first to second layer) is followed by the input-parallel computation (second to third layer), the second layer results do not need to be explicitly serialized (in a DNN, the second layer activations are

only "intermediate activations" used only by the third layer). Based on this observation, we developed a processing-in-memory module (PIM) that can house a 3-layer binary neural network with maximum layer widths of L–H–L as a unit (Fig. 4.6).

A PIM consists of two H-bit L-word SRAM arrays with output- and input-parallel neural engines (OPNE and IPNE). The OPNE sequentially receives the input activation bit stream A1, which is column-parallel XNORed with H-bit weights W1 read in parallel from the upper SRAM cycle by cycle and accumulated in real values. After the L cycle, the input activation traversal, accumulation, and activation (taking sign) are completed, and the resultant H-bit output activation vector A2 is stored in the registers (flipflops) between the OPNE and IPNE. Activation bit vector A2 is then fed in parallel into the IPNE, where column-parallel XNORs and an adder-tree process them by using the H-bit weight vector W2 read from the lower SRAM. This produces an output activation every cycle; the activation constitutes the bit-serial output activation stream A3. The OPNE and IPNE process weigh matrices in an orthogonally transposed manner, that is, a weight vector used by an output neuron is stored vertically in the upper SRAM and horizontally in the lower, respectively (in other words, a word of the upper SRAM is the weight *from* an input neuron, while that of the lower SRAM is the weight *to* an output neuron). By using wide SRAMs, long weight/activation vectors can be processed at once, while restricting the input/output of a PIM to a single bit.

4.4 Full Architecture (Ternarized/Biased) and Prototype Chip

Recent studies have shown that compared to binary DNNs, ternary DNNs (whose weights take values +1, 0, or −1) achieve higher accuracy in several situations [49], as mentioned in Section 4.2, and as discussed in Section 4.6. In addition, batch normalization techniques [37] are indispensable in binary/ternary neural networks to keep the weights informative under extreme approximation and obtain higher accuracy [18, 68].

In this study, we extended the base architecture by introducing mask (M) and bias (B) bits to accommodate ternary weights and batch normalization, aiming at handling versatile DNNs by using a single reconfigurable architecture.

4.4.1 Ternarized Neural Network

We introduce the mask (M) bits into the base computation model mentioned in the previous section to allow ternarized DNNs to be mapped onto it. When the mask (M) bit is enabled, the corresponding weight is treated as value "0" (note that logic 0 in the weight bits represents value "−1"; thus, the original binary DNN algorithm does not have any way to represent zero-valued synapse).

Moreover, this extension makes the PIM versatile and reconfigurable because zero-valued weights are equivalent to "non-existing synapses." There are some algorithms called "pruning" [35], which eliminate some powerless synapses/neurons in a neural network layer. CNNs can also be seen as a fully-connected neural network with sparse and repeating synapse weights. By setting nonexisting synapse to a zero value, we can map the neural network layer with an arbitrary shape (dense or sparse, wide or narrow) under maximum width.

4.4.2 Bias Terms and Batch Normalization

The bias term (Fig. 4.5) is necessary, especially for emulating batch normalization in binary/ternary neural networks. Batch normalization [37] normalizes the distribution of the activations in each layer statistically, thus speeding up the training and improving the generalization ability of the network. Batch normalization is defined by the following formula [37]:

$$\hat{x} = \gamma \left(\frac{x - \mu}{\sigma} \right) + \beta \tag{4.4.1}$$

where x is the sum (*i.e.* $\sum Wa$ in the previous explanation), μ and σ^2 are respectively the mean and variance of x over all the input images (or vectors) in the training set, and γ and β are respectively the trainable scaling and offset factors. Then, an output in the binary/ternary neural networks is obtained as

$$\begin{aligned} a' &= \text{sign}\left(\hat{x}\right) && (4.4.2) \\ &= \text{sign}\left\{ \gamma \left(\frac{x - \mu}{\sigma} \right) + \beta \right\} && (4.4.3) \end{aligned}$$

According to Yonekawa *et al.* [89], since binary/ternary networks require only the sign, this equation can be modified into

$$a' = \text{sign}\left\{ x + \left(-\mu + \frac{\sigma}{\gamma}\beta \right) \right\} \tag{4.4.4}$$

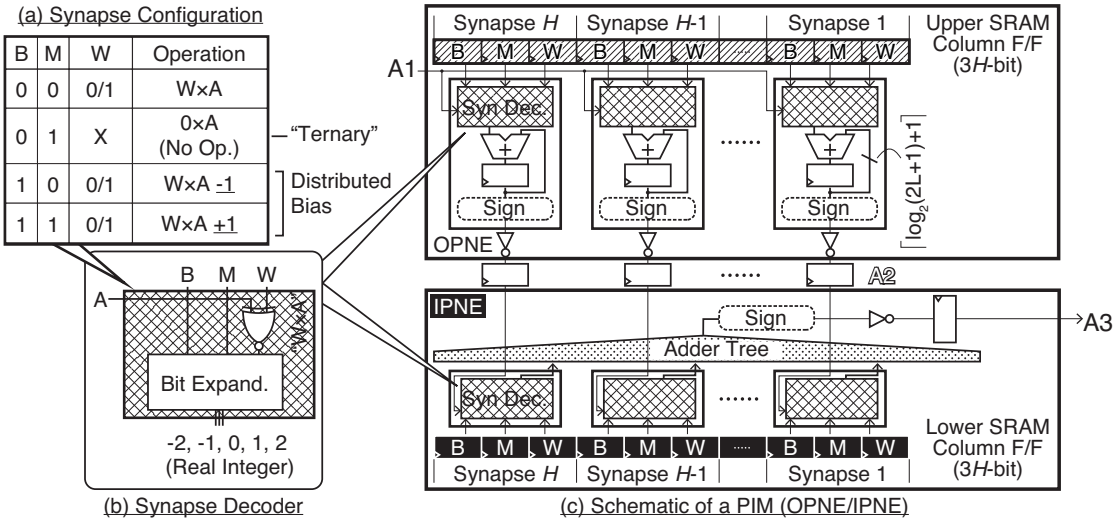

Fig. 4.7: Synapse decoder (SynDec) and detailed OPNE/IPNE circuit for extended ternary neural networks and bias terms.

because γ and σ are positive constants in the inference phase. This implies that all the batch normalization constants can be packed into a bias constant $\left(-\mu + \frac{\sigma}{\gamma}\beta\right)$ after the completion of the training phase [89].

To represent the real-valued constant bias terms in addition to the weight values, we introduced the "bias" B bits. The bias is coded into B bits in a distributed manner, that is, the absolute value of the bias term of an output neuron is the number of B bits enabled in its input weights, with its sign coded by the M bits (the M bits are diverted as the signs of the bias values when the B bits are asserted). The bias values are added or subtracted to the bit count in the OPNE/IPNE. By accommodating this bias addition mechanism, our architecture can apply batch normalization without needing additional multipliers/dividers to calculate normalized activations. Note that we adopted this distributed bias expression because of the symmetry of the OPNE and IPNE, thus overlooking the loss of the data representation density on the SRAMs. A bias term is needed *per column* in the OPNE because the weights for an output neuron are represented in a column, while it is required *per row* in the IPNE because an output is represented in a row. Nonetheless, the distributed bias representation can be utilized equally in the upper and lower SRAMs.

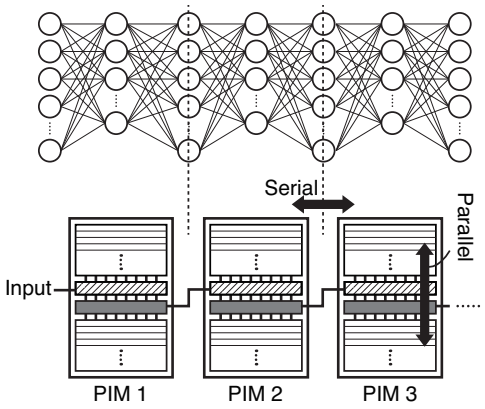

Fig. 4.8: Processing multilayer neural network onto the cascaded PIMs.

4.4.3 Architecture Design with Ternary and Bias Extension

All the possible configurations of the extended M and B bits in a weight (synapse) are enumerated in Fig. 4.7(a). Figs. 4.7(b) and (c) show the detailed schematics of the OPNE and IPNE that consider the ternary and bias extensions, respectively. We define a small logic unit as "synapse decoder" (or *SynDec*), which determines the real number (partial sum) to add in the accumulator or adder tree according to the weight W, mask M, bias B, and input activation A bits.

The OPNE is composed of arrayed H SynDecs, followed by accumulators; it computes H "ternarized" multiplications by using one input activation A1 and H accumulations in L clock cycles, and then take the signs to produce the activation after summing up all the partial sums. The IPNE also consists of the SynDecs; its main difference to the OPNE is the existence of an adder tree. Here, the partial sums are summed up in an adder tree and an output activation A3 is generated in a clock cycle. The OPNE/IPNE and two SRAMs are packed to form a PIM. A PIM conducts bit-parallel computation of two output layers of a neural network by using the bit-serial input and output.

4.4.4 Macro- and Micro-Pipelined Execution and Reconfigurable Array Extension

Connecting PIMs inline is a straightforward extension. Since the input and output of a PIM are 1-bit serial streams, multiple PIMs can be easily cascaded for deeper neural network processing, thus keeping the computation inside a PIM parallel (Fig. 4.8). Each of the OPNE and IPNE processes a layer; thus, two layers are hosted by a PIM.

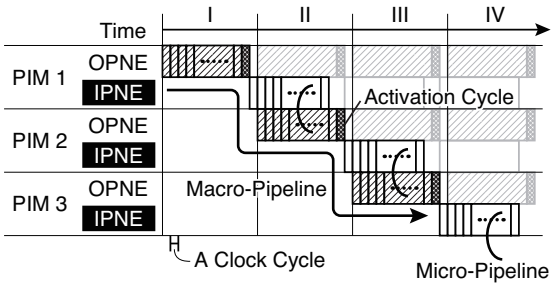

Fig. 4.9: Macro- and micro-pipelined computation on the cascaded PIMs. An "activation cycle" at the end of every macro-pipeline stage is required for taking the sign of the accumulator values and for issuing the SRAM addresses.

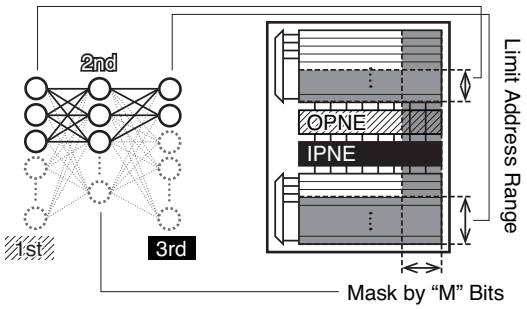

Fig. 4.10: Mapping a smaller layer onto a PIM. We can use smaller layers by limiting the address range (for the input and output activations) and by masking unused weights (for the intermediate layer).

The input bit stream is processed by OPNEs/IPNEs in a macro- and micro-pipeline parallel manner (Fig. 4.9) in addition to the internal column-parallel processing.

The output of the OPNE is calculated and determined after $(L + 1)$ cycles (all L input activations pass and the sign bit is taken), then the IPNE functions by using this OPNE result (stored in pipeline registers). While the IPNE is computing, the OPNE can start the processing of the next input frame ("transaction"). We call this a "macro-pipeline," since the activations move *every* $(L + 1)$ *cycles*.

The output of the IPNE is a single bit that is generated every clock cycle. This output is used immediately as the OPNE serial input at the next PIM. We call this a "micro-pipeline," because the activations flow in the processing engines *every cycle*.

A PIM can compute layers smaller than the maximum size L–H–L, as shown in Fig. 4.10.

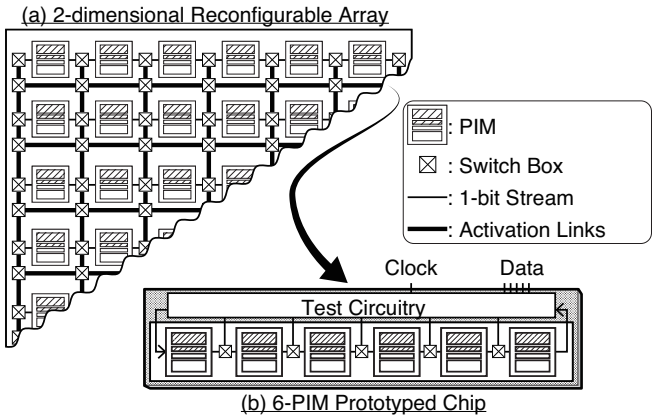

Fig. 4.11: Reconfigurable 2-dimensional PIM array and prototype chip with inline-arranged 6 PIMs.

For the first layer (serial input) or the third layer (serial output), the computation can be performed by limiting the address range of the weights in SRAMs. For the second layer (intermediate result between the OPNE and IPNE), we can "mask" the surplus synapses to prevent those unused values from being summed.

By arranging PIMs in a reconfigurable regular array, the architecture becomes transformable and expandable (Fig. 4.11). Here, activation links among PIMs are bundled bit-wise signals and are circuit-switched at the switch boxes (like in FPGA) so that the activations can be looped back (for RNNs), duplicated (for wider DNNs), or chained (for deeper DNNs).

4.5 Experimental Results and Comparisons

4.5.1 Prototyped Chip Evaluation

We prototyped a portion of the reconfigurable array by using the TSMC 65 nm GP process (Fig. 4.12). SRAM sizes L and H were set to 484 and 144, respectively, based on the availability of an SRAM library. We successfully integrated 6 PIMs capable of hosting a maximum 13-layer, 4.2K-neuron/0.8M-synapse binary/ternary DNN in 3.9 mm^2. In every clock cycle, 3,456 operations ($144 \times 2[\text{OPNE} + \text{IPNE}] \times 6[\text{PIMs}] \times 2[\text{Add} + \text{Mul}]$) are conducted in parallel. The chip operates at 400 MHz, achieving 1.4 TOPS (tera operations per second; 1 synapse MAC = 2 operations) with 0.6 W.

We measured the power consumption and critical path delay on several supply voltages

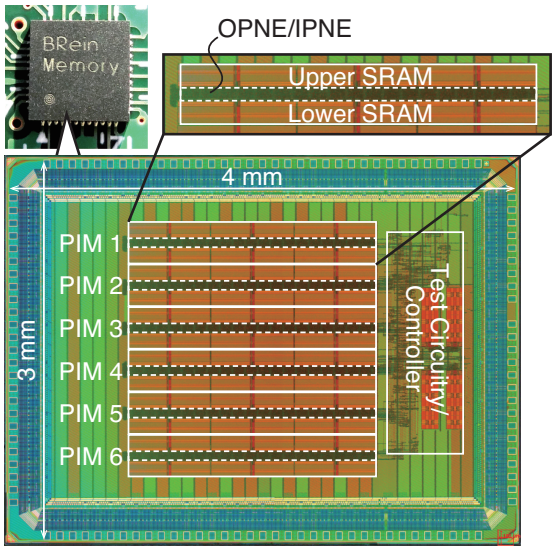

Fig. 4.12: Prototyped chip.

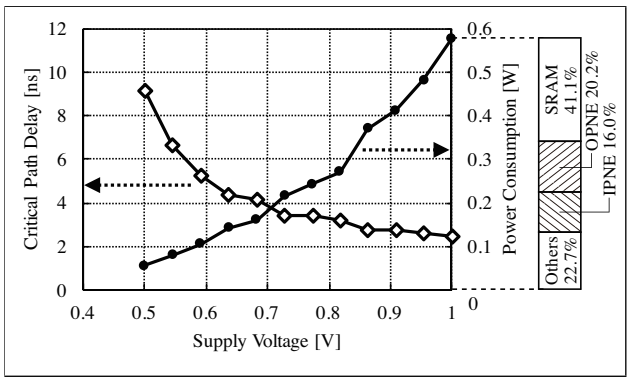

Fig. 4.13: Critical path delay and power consumption over the supply voltage. The chip achieves 400 MHz and 0.58 W at 1.0 V power supply. The critical path of the architecture is the path from the OPNE output register to the IPNE output register. The SRAMs (with the flip-flops), OPNEs, and IPNEs account for 41.1%, 20.2%, and 16.0% of the total power consumption, respectively.

(Fig. 4.13). The chip works at the range from 110 MHz (min. 50 MHz) at 0.55 V supply to 400 MHz at 1.0 V supply, with a power consumption from 0.06 to 0.58 W. The critical path of this architecture is in the adder tree in the IPNE, that is, the path from the OPNE output registers to the IPNE output register, where the sum of the L intermediate products and

TABLE 4.1: COMPARISON ON THE SAME 13-LAYER BINARY DNN (MEASURED)

	Process Tech.	Clock Freq. [GHz]	Exec. Time [sec]	Power [W]	Energy Consum. [J]	Effect. Perf. [GOPS]	Energy Efficiency [GOPS/W]	Ratio			
								Clock Freq.	Exec. Time	Power	Energy Consum.
CPU	14nm	2.2	124	155	19.2k	12.4	0.08	5.5	102	266	27.1k
GPU	28nm	1.0	13.8	152	2,098	111.3	0.73	2.5	11	261	2,966
FPGA	28nm	0.2	53	11	583	29.0	2.64	0.5	44	19	825
This Work	65nm	0.4	1.22	0.6	0.73	1264.4	2172.42	1	1	1	1

Measured using 1 million transactions of the binary DNN handwritten digit recognition. We assumed that 1 MAC is composed of 1 ADD and 1 MUL; thus, 1 MAC is equivalent to 2 OPs.

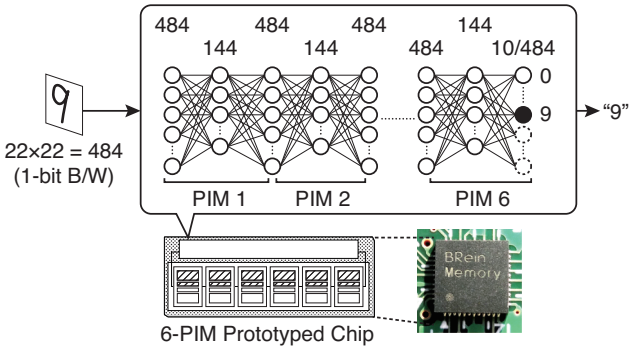

Fig. 4.14: Testing handwritten digit recognition on a 13-layer fully-connected binary multilayer perceptron (MLP).

the sign of the sum are calculated within a cycle. When we scale up the architecture, the maximum clock frequency will be limited by the width (*i.e.* the depth of the adder tree) of the IPNE. The ratio of the power consumption of the SRAMs (including the readout flip-flops), OPNEs, and IPNEs to the total are 41.1%, 20.2%, and 16.0%, respectively.

4.5.2 Comparison with CPU, GPU, and FPGA

We tested the proposed chip through an experimental handwritten digit recognition by using a fully-connected binary DNN (Fig. 4.14). To measure the core power consumption

and execution time, we prepared a 13-layer binary DNN, which occupies all the PIM computational resources and the SRAM area, except the last layer (that is, output layer; only 10 neurons corresponding to the classes are enabled). We trained the 13-layer DNN by using the trainer software we developed using Chainer and TensorFlow™; this is explained in the next section. The training dataset is not MNIST-compatible, but is generated from the MNIST dataset by resizing the input images to 22×22 (because of the limit of the SRAM depth 484, which corresponds to the input data size), and binarizing these images into black-and-white images through thresholding. The recognition accuracy reached 90.1% at 13 layers (details will be explained in Section 4.6).

When compared with a CPU∗, GPU†, and FPGA‡ running the same binary DNN, the chip achieved 1-–2 and 2—4 orders of magnitude better performance and energy efficiency, respectively, with the 0.5x–5.5x lower clock frequency and 1–2 orders lower power consumption (Table 4.1). In addition, we examined a binary DNN inverted pendulum robot and game player through reinforcement learning.

4.5.3 Comparison with Prior Accelerators

Recent hardwired binary neural network implementations [63, 60] are custom solutions with insufficient versatility in their scope. The comparison of recent state-of-the-art CNN accelerators [17, 73, 55] with this work (Table 4.5.1) is not straightforward: they are nonbinary (vs. binary) and are built for being used in the CNN (vs. targeted for a wide variety of DNNs). For the metrics in Table 4.5.1, the former (latter) performs positively (negatively) to the proposed method. With this precaution, the peak performance [TOPS] of our chip outperformed that of the prior studies. The energy efficiency [TOPS/W] and area efficiency [TOPS/mm^2] were in better or comparable range, considering our older CMOS process technology (we also estimated the 65nm-normalized power and area efficiency of the prior works). This is mainly because the wide SRAMs cost the 66% area and 41% power, respectively. However, considering that the prior CNN accelerators access off-chip memories whose area and power are not included, the balance is actually

∗The test environment is a CentOS 6.4 running on a 128 GB DDR4 main memory and two CPUs, each of which is Intel™ Xeon® E5-2650v4, 2.20 GHz, 12 cores, with 30 M cache. The vector computation library was not enabled.

†The GPU we used is NVIDIA™ GeForce® GTX TITAN X, which has 3,072 cores and a 12 GB VRAM, installed on the same host system used for the CPU evaluation.

‡The FPGA is Xilinx™ Virtex®-7 XC7V690T FFG1761-3 mounted on the Digilent® NetFPGA-SUME board with 8 GB DDR3 and 216 Mbit SRAM. The board is installed on the same system used for the CPU evaluation.

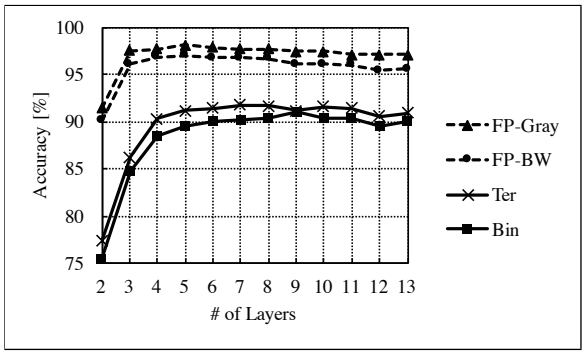

Fig. 4.15: Accuracy behavior when the number of layers varies (including the input layer). "Ter" and "Bin" are the results of ternarized and binarized weights, respectively, which can be mapped to the prototyped chip, while "FP-Gray" and "FP-BW" are grayscale and black-and-white images, respectively, used for comparison. The sizes of intermediate layers (the layers except for the input and output layers) are 484–144 and the input image is 22×22 pixel black-and-white due to the SRAM width of the prototyped chip.

much more favorable to the proposal.

4.6 Training Environment and Algorithm Evaluation

We built the neural network trainer and emulator software tuned to BRein Memory for three purposes: 1) to examine the effectiveness of the refined BRein Memory architecture, 2) to run application evaluation awareness of the hardware features, and 3) to consider and determine the requirements of the future architecture. We developed the custom operation library of Google TensorFlow™ by using the aforementioned extensions, such as weight ternarization and bias-term-based batch normalization. We used the customized TensorFlow™ framework in the construction of the application described in this section.

4.6.1 Application Test Using the MNIST Handwritten Digit Dataset

First, we implemented the handwritten digit recognition on the MLP that we mentioned in Section 4.5.2 with three types of data representations, that is, binary, ternary, and floating point, for the comparison. The flexibility in the depth (number of layers) of the neural network is one of the features of our architecture; thus we conducted a parameter exploration to show the flexibility on the layer number, and to determine the optimal layer number. We trained the MLPs with different numbers of layers by using their shapes, which

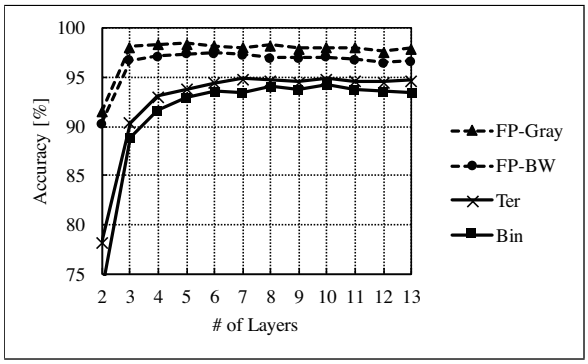

Fig. 4.16: Accuracy behavior when the number of layers varies (including the input layer) in the case the SRAMs are wider (having more bits per word) than that in the prototype chip. The intermediate layer sizes are fixed to 1,024 and the input image is 22×22 pixel black-and-white.

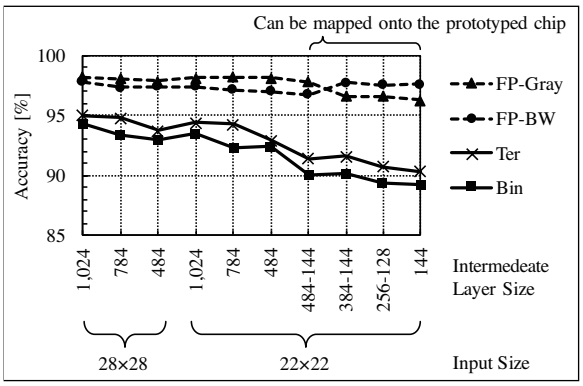

Fig. 4.17: Accuracy behavior when the intermediate layer width varies.

agree well with the current PIM design (484–144 style). The input images were resized to 22×22 (=484) and binarized through thresholding. The result of the layer-number sweep is shown in Fig. 4.15, where "Ter" is a ternalized network by fixed thresholds; "Bin" is a binary network, which does not allow zero-valued weights; and "FP-BW" and "FP-Gray" are 32-bit floating-point weights including and excluding the input binarization, respectively. The accuracy increases and saturates when the layer number scales. For this task, a 6-layer (or 5-layer if counting the input layer out) MLP seemed to be sufficient.

We mentioned that the 484–144 style OPNEs/IPNEs are only used for chip prototyping and the architecture itself could be utilized with a larger/smaller memory. To observe the behavior of the case in which the layer sizes were uniform, we tested the same

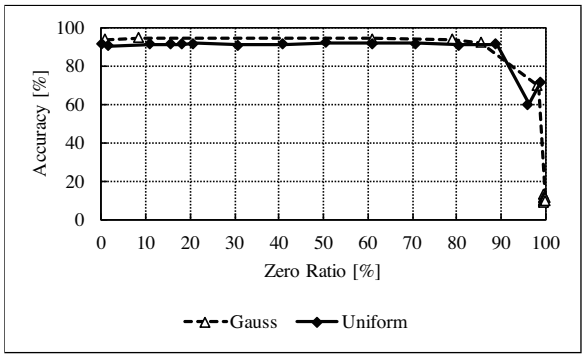

Fig. 4.18: Accuracy depends on the zero-ratio of the total "ternarized" parameters. Over 90% accuracy is obtained with 85% weights are zero. "Gauss" and "Uniform" are the initial distributions of the weights.

handwritten digit recognition task by using an MLP with 1,024 neurons per intermediate layer. Although this assumption is above the capacity of the test chip, it could be realized if we utilized a $(1,024 \times 3)$-bit 1,024-word SRAM macro. For comparison, the input size was fixed at 22×22. According to Fig. 4.16, the accuracy behavior when the number of layers varies is quite similar to that of the 484–144 case, with an improvement in the final accuracy. A 6-layer MLP is still sufficient for this extended layer size assumption.

The two above-mentioned parameter sweeps have shown that a 6-layer MLP is sufficient both in the current 484–144 test chip and the extended 1,024-neuron model. Next, we tested the effect of the layer width with the layer number fixed to 6. Fig. 4.17 indicates that the recognition accuracy still improves when we utilize richer layer/input width configurations; thus, there is room for employing wider OPNEs/IPNEs paired with wider SRAMs. Though the accuracy of the binary/ternary networks in this test is a few percent lower than that of FP32 networks, there are some applications where the efficiency, throughput, and the shorter response time are much more important than a slightly higher accuracy, such as mobile or embedded always-on security systems.

4.6.2 Threshold and Zero-Ratio

The ratio of zero-valued weights can be controlled by simply setting the thresholds. Fig. 4.18 indicates the relationship between the accuracy and zero-ratio of a handwritten digit recognition MLP. We tested two methods for parameter initialization at the beginning of the training phase: uniform and Gaussian distributions. In our small application model, the difference between the initial value affected the number of training steps (or

"epochs") and the final distribution (and zero-ratio) of the weights. However, the effects of zero-ratio on the accuracy are similar. This result shows that over 90% accuracy is maintained even if 85% of the weights are valued zero in this task. This suggests an opportunity for data compression (or *pruning*), that is, reducing memory requirement by eliminating zero-valued meaningless weights. Although our current architecture cannot exploit data sparsity, this observation, expecting network compression, would be key in future reconfigurable architectures.

4.6.3 Experimental CNN Mapping

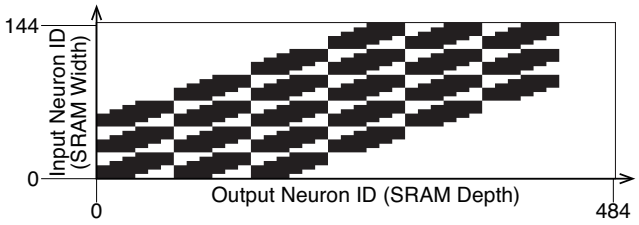

Fig. 4.19: Writing down a convolutional layer as a fully-connected layer (used the second layer of Table 4.6.3 sized $6 \times 6 \times 12$ as an example). The horizontal axis is the addresses, and the vertical axis is the bits of the SRAM (acctually the addresses are tripled for three-bit synapse representation). The black pixels are used as kernel weights, and the white area is filled with 0s. Since a convolutional layer applies a single kernel to multiple output neurons, many of the black blocks are duplicated.

The proposed architecture is not optimized for CNNs; however, it has the capability of housing a CNN by representing a convolutional layer as a fully connected layer. The maximum configuration of a convolutional layer is only limited by the layer size (*i.e.* the sizes and numbers of feature maps). The kernels are statically preprocessed and reshaped into fully connected forms, with zero-valued weights representing the outside of the receptive field of a certain output neuron. As indicated in Fig. 4.19, this method enables us to map a convolutional layer to our PIMs but the efficiency may be limited; a considerable number of "zero-valued" synapses account for memory capacity, and some regularly duplicated values arise from the same kernel but different input/output neurons.

We conducted an experiment of a simple CNN mapping onto the prototyped 484–144 chip, with the same MNIST digit recognition task. To map a CNN onto the experimental

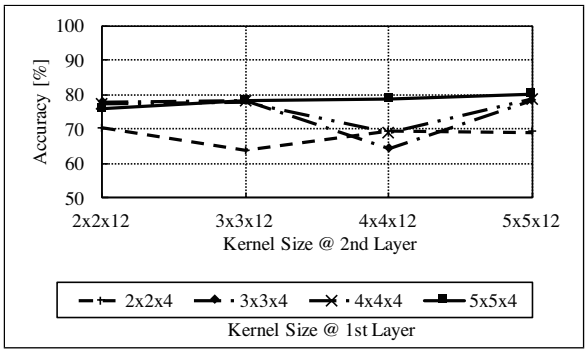

Fig. 4.20: Accuracy of the first and second layers over the kernel sizes. The horizontal axis is the first-layer kernel size, and each line corresponds to the second-layer kernel.

chip, we chose the layer configurations shown in the Table 4.6.3 ($K_h \times K_w \times K_c$ implies a kernel shaped $K_h \times K_w$ with K_c output channels). The highest accuracy was observed at $5\times5\times4$ for the first layer and $5\times5\times12$ for the second layer; however, it resulted in a lower accuracy than the fully-connected evaluation. This is mainly because of the resized input image and narrow intermediate layer due to the narrow SRAM width of the prototyped chip. The accuracy will improve if we use a wide SRAM with wide OPNE/IPNE that can store many channels of the feature map.

4.7 Conclusion

In this chapter, we proposed an efficient near-memory accelerator architecture for binary/ternary neural networks. The architecture enabled efficient highly-parallel computation near the SRAMs under the serial input/output data, exploiting the parallelism that the neural networks originally have. By arranging the serial inputs/outputs of processing units and configuring the synapse weights on the SRAMs, this architecture has the versatility to host any type of network with any depth, such as fully connected DNNs, recurrent DNNs, and sparsely connected DNNs (CNNs and pruned DNNs), in a reconfigurable manner.

We fabricated the prototyped chip on 65nm CMOS with twelve 484-word (144×3)-bit SRAMs and examined the architecture's availability through an application test by using our developed training/testing environment. The chip measurement and comparison proved that the energy and area efficiencies achieved 2.3 TOPS/W and 0.089 TOPS/mm^2 respectively, which were higher or comparable to the highly-dedicated accelerators at the time of publication. In parallel to keenly optimized CNN accelerators, such a versatile DNN accelerator will become prevalent as algorithms and applications for DNNs continue

to evolve.

The application test also indicated the issues to be solved. Although the sparse or convolutional neural network models can be mapped onto the PIM array, the computational efficiency degrades compared with fully-connected dense ones. To further improve efficiency, native support is needed to map the sparse models onto SRAMs and skipping the zero-weighted multiplication.

TABLE 4.2: COMPARISON AND CHIP SUMMARY

	ISSCC 2016 [17]	**ISSCC 2017 [73]**	**ISSCC 2017 [55]**	**This Work**
Technology	65nm LP	65nm	28nm FD-SOI	65nm GP
Target	CNN	CNN + RNN	CNN	Versatile DNN[1]
W/A Precision [bits]	16	4–16 (Conv), 4–7 (FC)	4–16	Bin/Ter
Operating Frequency [MHz]	200	50–200	200 (Typ.)	100–400
Voltage [V]	1.0	0.77–1.1	0.6–1.1	0.55–1.0
Power [W]	0.3	0.03–0.28	0.075–0.3	0.05–0.6
Core Area [mm^2]	12.3	7.4	0.95	3.9
Peak Performance[2] [TOPS]	0.07	1.25	0.41	1.38
Energy Efficiency [TOPS/W]	0.2	1.0–8.1	0.26–10 [0.05–1.86][3]	2.3–6.0
Area Efficiency [TOPS/mm^2]	0.006	0.045–0.169	0.108–0.431 [0.02–0.08][3]	0.089–0.365
Dataset (Network) for Evaluation	(AlexNet)	(AlexNet)	Hierarchical Face Recognition (AlexNet, VGG-16)	Resized MNIST (Binary/Ternary MLP)

[1] Although our architecture is not specially optimized for CNNs, it can still house CNNs by mapping convolutional layers as fully connected layers, as mentioned in Section 4.6.3.

[2] Note that the definition of OP differs chip-wise. We estimated the performance by assuming that 1 MAC for 1 synapse is composed of 2 operations (addition and multiplication).

[3] The efficiency assuming the system is integrated on a 65nm technology, that is calculated using the scaling law (Power, Area $\propto$ (Gate Length)^2).

Table 4.3: Layer Configurations of the Experimental MNIST CNN

#	Type	Kernel	Output
0	Input	–	$22 \times 22 \times 1$
1	Conv	$2 \times 2 \times 4$, $3 \times 3 \times 4$, $4 \times 4 \times 4$, or $5 \times 5 \times 4$ with Stride 4	$6 \times 6 \times 4$
2	Conv	$2 \times 2 \times 12$, $3 \times 3 \times 12$, $4 \times 4 \times 12$, or $5 \times 5 \times 12$	$6 \times 6 \times 12$
3	F.C.	(F.C. layer from 432(= $6 \times 6 \times 12$) to 144)	144
4	F.C.	(F.C. layer from 144 to 10)	10

Chapter 5

Dither NN: Accurate and Efficient Quantization Algorithm Enabled by Hardware-Software Co-Designing

5.1 Introduction

The computational and memory cost of neural network processing has become a problem, especially on embedded systems. We constructed an energy-efficient architecture by adopting binary neural networks in Chapter 4 to address it. Quantization algorithms reduce the computational complexity, thus improves the performance and efficiency of neural network processing. It has become one of the most important research topics in neural networks, as the application demands of edge-side user-side AI processing are expanding. However, the accuracy degradation in a quantized neural network model due to poorer information representation is unavoidable. A light-weight but accurate quantization algorithm is desired.

In this chapter, we consider a quantization algorithm starting from the computational structure of neural network hardware in an algorithm-hardware co-optimization fashion. Inspired by the quantization algorithms that have been researched and utilized in the field of signal processing, we introduce dithering to the quantized neural network algorithm. The required calculation for dithering can be done using the structure that MAC-array-based neural network processors originally have, for which no additional arithmetic units are needed. Therefore, this algorithm does not degrade the efficiency of neural network processor with quantization.

This work is based on our previous conference and jornal papers [6, 7]. The rest of this chapter is composed as follows. In Section 5.2, we take a brief look at the prior work on

hardware-oriented low-precision neural networks and their accelerator architectures. In Section 5.3, the base algorithm of the dithering in the field of digital signal processing is explained. Then in Section 5.4, the algorithm is modified and extended to be used with neural network hardware. In Section 5.5, we discuss applying the backpropagation training method to the proposed dithering neural network, and some additional dithering techniques are discussed. In Section 5.6, we conduct the accuracy evaluation using software simulation, and we evaluate the hardware efficiency of the proposed method by FPGA prototyping. Section 5.7 discusses a conceptual proposal of using the dithering in generic low-precision hardware other than the binary one. Finally, we conclude the chapter in Section 5.8.

5.2 Related Work

Multiplication is the costliest computation of the neural network, and the memory occupation is caused mostly for the weights. Multiplication requires a lot of hardware resource; the gate count is in proportion to the square of the bit width, in contrast to addition whose gate count is in linear proportion. Data amount, on the other hand, gets larger as the network model becomes more complex, owing to the demands on the practical applications. To reduce these computation and memory resource requirements, mainly for mobile applications that cannot hire large memory and energy-hungry high-end processors, many hardware-aware approximate neural network algorithms have been proposed.

The redundancy of a neural network model has been researched since the dawn of the deep learning applications. It is known that a neural network does not require much higher numerical precision, especially in the inference phase. Studies have been conducted that use fixed-point expression rather than floating point, aiming at simplifying the computation (both multiplication and addition) [19, 28]. An approach called *dynamic fixed-point* that allows the bit width in a single network to vary has been proposed [54, 56, 73]. In the most aggressive case, the bit precision alternates during the inference on an accelerator according to the occurrence of the latest arithmetic overflow/underflow.

BinaryConnect [20] appeared in the second half of 2015 as the first successful neural network model that uses binary weights. It *binarizes* the weights into ± 1 while keeping the activations linear, which allows the maltiply-accumulation (MAC) operation between an activation and a weight to be calculated as an addition. During the training phase, the full-precision weights must be kept since the update of the weights takes place in full-precision as a part of back propagation. When the training phase is complete, the weights are statically binarized, and in the inference phase only the binarized weights are

retained.

The first feasible network model that binarizes both the weights and the activations was *BinaryNet* [18] in 2016, which is also called binarized neural network or binary neural network (BNN). Its binarized weights and activations can be *multiplied* by a bit-wise XNOR (exclusive NOR) operation; therefore, the heavy multiplication step can be eliminated. The success of this model is attributed to "batch normalization" [37]. Batch normalization regularizes the distribution of the activations statistically before binarizing them, which tolerates the statistical changes in the input activations among batches in the training phase; thus preventing the network from overfitting/diverging and accelerating the training.

These binary network models often result in lower recognition accuracy due to their extreme approximation. There are several approaches that use *quantized* numerical expression other than binarization. Ternary weight network [49], or ternary neural network, quantizes the weights into ternary expression with the values of -1, 0, and $+1$, which could achieve a more accurate weight representation than the binary one in a certain situation. *LogNet* [53] quantizes the weights and activations in the logarithmic representation, which replaces the costly multiplication with a simple addition while retaining the resolution of the most frequently occurring numbers valued at around 0.

XNOR-Net [68] appeared just after BinaryNet. It conducts the binary weight-activation XNOR multiplication with a few real-valued scaling factors. Residual binary neural network (*ReBNet*) [26] is an extension of the binary neural network that binarizes both the activation and weights, intended to obtain the appropriate approximation by a linear combination of the binary activations. It binarizes the activations gradually into a sequence of binary numbers using multiple thresholds and "residual errors". The resulting multiple binary activations are multiplied (XNORed) with a single binary weight, which are then linearly combined using the scaling factors.

5.3 Dithering

In this section, we explain the base algorithm of "dithering" to accommodate to neural networks.

5.3.1 Dither in Signal Processing

Dithering is a commonly-used technique in digital signal processing, used to reduce the effect of quantization errors. When a source signal is "quantized" into another signal

digitally, quantization errors (the differences between the source and quantized signal) always occur. For example, when a photograph in 24-bit color is converted to 8-bit color, the details and gradients will be lost and parts that had similar colors may become indistinguishable.

Dithering is often used with binary quantization where the quantized signal only takes two states (in 1-bit digital signal, they are '0' and '1'). In binary quantization, a threshold is set, and an input value is quantized into '1' when it is greater than the threshold; otherwise it becomes '0'. This means that if many of the input samples have values slightly lower than threshold, they always become '0', and as a result, the total quantization error becomes quite large. A solution is to stochastically quantize nearly half of those middle-valued samples into '0' and the other half into '1'. This would equalize the quantization errors among the input samples so that the total quantization error becomes minimal because half of the output values will have negative errors (deficit of the output), while the others will have positive errors (surplus of the output). This idea of stochastically quantizing the inputs is called "dithering."

5.3.2 Error Diffusion

One of the most widely used approaches of dithering is the "error diffusion" method. In the following explanation, consider that the N-length source signal S takes a real value in $[0, 1]$ at the i-th sample S_i for each i ($i = 1, \cdots, N$), the resulting quantized signal Q takes a binary value (0 or 1) at the i-th sample Q_i, and the threshold is T ($0 < T < 1$). Note that the error diffusion algorithm does not limit the output signal to be binary; it can be utilized with any kind of quantization; however, we use binary quantization as an example here. As shown in Fig. 5.1 and Algorithm 1, the basic idea of this method is to "integrate" (or "accumulate") the quantization error occurring at each sample. The current quantization error E (surplus/deficit of the quantized value when it is 1/0) is added to the next source value S_{i+1} before thresholding. This works to cancel the total (or average) quantization error among overall samples as described above. This algorithm is quite similar to the "delta-sigma modulation" that is used in ADCs (analog-to-digital converters).

Let us give an example. If the first input is $S_1 = 0.3$, the second is $S_2 = 0.4$, and the threshold is $T = 0.5$, the quantized values without dithering would be $Q_1 = Q_2 = 0$ because both inputs do not exceed the threshold. In this case, the total quantization error is $\sum(S_i - Q_i) = 0.7$. When we apply the error diffusion method to these inputs, the first quantized output is $Q_1 = 0$ since the initial value of the integrated quantized error E is 0; therefore, the error becomes $E = 0.3 - 0 = 0.3$. Then, this error $E = 0.3$ is added

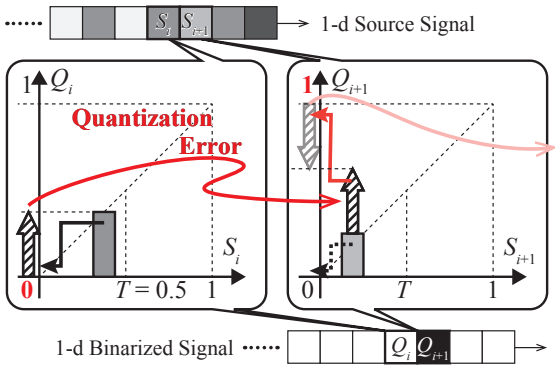

Fig. 5.1: Error diffusion on 1-dimensional signal quantization.

Algorithm 1 binary-quantization of 1-d signal using error diffusion

Input: $S = \{S_1, \cdots, S_N\}$:
N-length sequence of a source signal (analog or digital)
Input: T: Threshold
Output: $Q = \{Q_1, \cdots, Q_N\}$:
N-length sequence of the binary-quantized signal

$E \Leftarrow 0$
for i in $\{1, \cdots, N\}$ **do**
  if $(S_i + E) \geq T$ **then**
    $Q_i \Leftarrow 1$
  else
    $Q_i \Leftarrow 0$
  end if
  $E \Leftarrow (S_i + E) - Q_i$
end for
return Q

to the next input ($S_2 + E = 0.4 + 0.3 = 0.7$), and the resulting output is $Q_2 = 1$ because it is greater than the threshold T. With the integrated quantization error, even a smaller input could produce an output 1, and vice versa. The total quantization error is now $(0.3 - 0) + (0.4 - 1) = -0.3$, whose absolute value is smaller than 0.7 of the case without dithering.

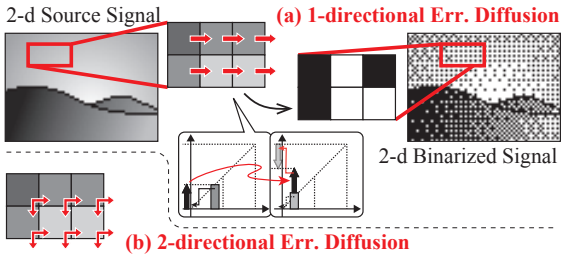

Fig. 5.2: Error diffusion on 2-d image. (a) "1-directional" method and (b) "2-directional" method.

5.3.3 Dithering on 2-d Signals

We have explained the error diffusion on a 1-d sequence, and this can be easily extended to 2-d signals (such as images) by propagating the quantization errors in the plane. The simplest way to achieve this is through a "1-directional" method that retains an integrated quantization error *for each row* as shown in Fig. 5.2(a), *i.e.* there is no such relationship between the rows. There are several algorithms used in image processing that *diffuse* the quantization errors of neighboring pixels (Fig. 5.2(b): an example of "2-directional" error diffusion). These multidirectional methods result in better image quality when used for image conversion, but the computational complexity would increase. We discuss these 2-d dithering algorithms later from the viewpoint of hardware implementation.

5.4 Neural Network with Dither

We have described the base algorithm of dithering, which minimizes the total quantization error for low-precision quantization. In this section, we attempt to apply it to neural network models to improve the accuracies of approximate neural network algorithms for mobile/embedded applications while minimizing additional requirement of hardware resources.

5.4.1 Prerequisite

Many convolutional neural network (CNN) based processors integrated on ASICs (application-specific ICs) and FPGAs feature output-parallel processing, where the processing engines (PEs) form an array and each PE computes an output neuron. On this parallel processor, each PE usually has a multiplier and an accumulator to calculate the MAC operations

between the inputs and weights, and the PE array sequentially scans all the input neurons while accumulating the products generated by the multipliers.

We assume this output-parallel processor as the main target of our work. As described later, this type of parallelism in which the output partial results stay in the accumulators and the inputs travel is suitable for the dithering algorithm.

5.4.2 Error Diffusion in Output-Parallel Binary Architecture

Baseline Binary Neural Network

We first discuss the general architecture with dithering for a binary neural network. Binary neural networks [18] restrict the activations and weights to either -1 or $+1$, unlike standard binary quantization used in signal processing, which uses values of 0 and 1. This is because the binary-quantized network is a special case of fixed-point (or linear integer) quantization where the bit width is 1, and therefore, only the sign bit remains. As proposed in [18], the multiplication between two binary-quantized values produces only four possible situations and can be executed as an XNOR (exclusive NOR) operation with the values $-1/+1$ being denoted as logic 0/1 respectively. A neuron of a binary neural network layer acquires the binary (± 1) pairs of weights and activations, conducts XNOR operation (*multiplication*) on each pair, sums all the resulting products, and picks the sign of the sum as the output activation (*non-linear function*).

To implement this logic on an output-parallel array, usually an XNOR-accumulator-based PE is used (Fig. 5.3(a)). The accumulator on a PE sums the *binary* products calculated by the XNOR gate from all the input activations in a *linear* manner.

Applying Error Diffusion

The summation of each neuron produces an integer value in the range of $-N$ to N for N input activations, and the resulting output activation is ± 1 depending on the sign. The quantization error can be defined in a similar manner to the basic explanation provided in Section 5.3.2, *i.e.* by simply subtracting the resulting activation ± 1 from the sum.

In the binary neuron processing on an output-parallel PE, the accumulator retains the final sum at the end of input scanning, and the value is reset to 0 immediately after the sign is assigned as the output. On the other hand, the quantization error to be added to the next neuron's weighted-sum in the dithering algorithm is calculated using the sum and sign of the current neuron. Thus, we can compute the accumulation of the quantization error by the following steps: 1) scan the inputs and perform XNOR-accumulation similar

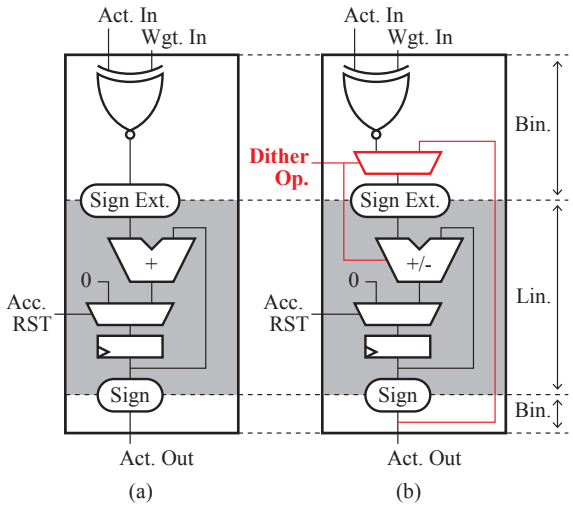

Fig. 5.3: (a) A general and simplified form of an XNOR-accumulator-type PE. (b) An XNOR-accumulator PE with error diffusion operation mode.

to the operation in standard binary neuron processing; 2) calculate the sign bit (the output activation) upon completion of the input scan; 3) subtract this ±1 value from the sum in the accumulator instead of the reset operation; and 4) scan the next input. Therefore, we do not need additional circuit components in the XNOR-accumulator-based PE; all that is required is a simple selector circuit to select the mode of the accumulator for the initial reset, accumulation, or quantization error calculation, as shown in Fig. 5.3(b).

The dependencies among the error diffusion sets in the output activations of the current neuron affect the next neuron; however, this would not be a problem for the output-parallel processing. A typical convolutional layer of a neural network has four axes in the output activation: batches, rows (height), columns (width), and channels (Fig. 5.4). We introduce the error diffusion technique only within a plane, *i.e.* the axes of rows and columns of each channel in each batch. Therefore, the axes of the batches and channels are unaffected. Most output-parallel architectures map the axes of the channels and rows onto their PE arrays mainly because of the efficiency of data delivery achieved. As long as we use the 1-directional error diffusion, the quantization error is only accumulated along a row; as each row/channel can be computed individually, the output-row-and-channel-parallel processing works well with dithering. In such a configuration, each PE in the array is assigned to an output neuron in a row of a channel and operates sequentially along the columns, as indicated in Fig. 5.5. In other words, error diffusion algorithms can be

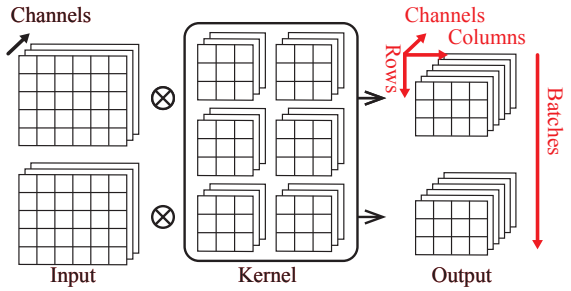

Fig. 5.4: Four axes of the activation in a convolutional layer.

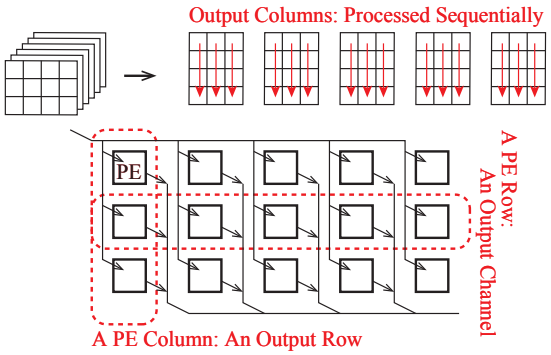

Fig. 5.5: Row- and channel-parallel PE. The output columns are processed sequentially, which harmonizes with dithering.

computed in a channel–row–column loop with the outer two axes unrolled spatially onto the output-parallel PE array. The output–input–channel parallelism is also suitable; in this case, the PEs in a column correspond to an output neuron, a row corresponds to an input, all the PEs individually calculate the MACs of the inputs and weights, and the partial sums finally obtained in the PEs are gathered and summed in a column-wise manner.

5.4.3 Combination with ReLU Activation Function

Activation Function in Conventional Models

A convolutional or fully-connected layer in a neural network employs a non-linear function called *activation function*. The activation function, which is applied to the weighted-sum of each neuron, is a key technique for neural networks. It represents a primitive non-linear

response by itself, and this non-linearity helps the network in obtaining an acceptable approximation of the complex behavior of an unknown implicit function.

Rectified linear unit (*ReLU*), is the most popular type of activation function in standard (linearly expressed) neural network models. This unit allows a positive value and blocks a negative value (set to 0). This lightweight implementation of a non-linearity requires checking only the sign of the input, and it is widely used because it is friendly with the training algorithms that use back propagation.

In a binary neural network, obtaining the sign of the sum is also a kind of activation function called *binarization* or simply *Sign function*. It can be interpreted as a thresholding function that checks if the weighted-sum exceeds 0 or not and can be generalized for any threshold value via a bias term.

Activation Function and Dithering

The output of the binarization using dithering approaches the identity transformation when the input is large enough and the output is "demodulated" by blurring it (equivalent to "integrating" or "applying low-pass filter"). This fact suggests that dithering does not have any non-linear effects. Therefore, we must consider introducing a mechanism for obtaining non-linearity in the error diffusion.

One possible option is to use dithering after a standard ReLU activation function. This is quite straight-forward and expects the ReLU activation function to provide the non-linearity and dithering to remap the result of the ReLU spatially. However, there are some disadvantages in this method: the computational complexity would increase owing to the separate operations for ReLU and dithering, and it may generate +1 in the neurons that the non-linear ReLU previously deactivated.

Another option is to customize the error diffusion algorithm, namely *positive-only error diffusion*, such that it takes the quantization error into consideration *only when the raw sum (S_i) is greater than the threshold*. Similar to the ReLU, the positive-only error diffusion suppresses a negative sum value to -1 unconditionally without using/accumulating the quantization error. When the sum has a positive value, it works like the baseline error diffusion, *i.e.* some of the positive inputs become -1 outputs by considering the quantization error or *total surplus*. This method could not only operate as a single activation function whose computational complexity would be reduced compared to the separated ReLU-dither structure but also properly evaluate the neurons having negative sums, which should be deactivated to obtain the non-linearity.

5.4.4 Working with Other Low-Precision Networks

The idea of dithering using error diffusion algorithm on a low-precision hardware can be naturally extended to any kind of quantization methods other than binarization, as long as the quantization error can be defined using the input and output of the quantization. We present basic concepts of using the dithering with the logarithmic and fixed-point quantization in 5.7, since they have not been synthesized and evaluated.

5.5 Discussion

5.5.1 Back Propagation Methods for Dither NN

When we intend to use a certain operation in the hidden layers of a neural network, its derivative *must* be defined for applying the back propagation. We discuss the appropriate realization of the derivative of the dithering operation.

The strictest and most accurate form of the derivative of the error diffusion is to mathematically differentiate it with no doubt. However, defining the strict derivative of the error diffusion algorithm is very difficult because it employs discontinuous non-linear operations, and because the computation of a neuron could depend on the results of many other neurons in the plane. Instead, we attempted to use another function to adopt back propagation into error diffusion.

Identity Function

First, we viewed the dither as an identity transformation. Since dithering aims to represent a richer expression by a poor quantized signal, the output must be similar to the input. The derivative of an identity function is simply the constant 1. This supposition worked well with some relatively small neural network models, but it was unstable. One reason for this failure is that the identity supposition is *too sensitive* (or *too much linear*) that many undesirable responses of the input/output activations (they are expected to be suppressed in a part of the non-linearity of the standard activation functions) affected the weights.

Active-Only Propagation

The basic principle of the back propagation is to claim the responsibility of the input for the output result. The idea is to pass the output error (or *delta*; not to be confused with the quantization error) only to the neuron that was activated as '+1' at the positive-

only dithering and activation, similarly to the back propagation for max pooling. This was totally unsuccessful, probably because it was the opposite of the case of identity assumption... it was *so dull* that the necessary feedbacks were trapped.

Partial Linear Approximation (Near-Threshold Propagation)

This method propagates the output error when the weighted-sum value before quantization is near the threshold and is very similar to the piecewise linear approximation of the Sign function of a standard binary neural network. It worked perfectly with the models we tested, from narrow and shallow networks to deep networks. We view the two characteristics of this setting: 1) the weights that must be updated waver near the threshold; 2) this near-threshold back propagation, where the neurons with the weighted-sum values in a certain range are updated, could harmonize with the characteristic of the dithering where the location of the neurons being activated is changed by the error diffusion.

Delayed-Linear Derivative

The same calculation as that for the error diffusion is used, as *delta-sigma modulation* in the field of digital audio. This is a form of ADC, which generates a pulse-density-modulated binary stream from the source analog signal. The transfer function of this modulation is denoted as $Y = z^{-1}X$, so the output sequence is linear to the input with 1-cycle delay. Based on this observation, we tested the *delayed-linear* derivative, where the output errors propagate linearly to the input with a 1-pixel shift. This implementation worked well in most of the experimental cases. We should select the near-threshold or this delayed-linear back propagation considering their computational load and non-linearity that would affect the training speed and final accuracy.

5.5.2 Complex Dithers

We mentioned that the error diffusion algorithm in image processing could be more complex for perceptually better image quality. This tends to enlarge the area for distribution of the quantization errors; for example, the Floyd–Steinberg dithering distributes the error of a pixel to 4 neighboring pixels. Many such algorithms distribute the error unevenly, *i.e.* they use multiple scaling factors for each direction.

However, we believe that this complex dithering can hardly be used with neural networks, because such processing procedures are too complicated to be integrated in an accelerator. The 1-directional error diffusion we experimented is the most friendly to the hardware

implementation. Moreover, we tested a uniform 4-directional error diffusion algorithm (distributes the error of a pixel to the right, lower-left, lower, and lower-right pixels with magnifying $1/4$) by software simulation, but the recognition accuracy did not improve as much as that in the case of the 1-directional version.

5.6 Evaluation

We evaluate the proposed algorithm from the viewpoints of accuracy and hardware cost. First, we conduct the simulation-based accuracy comparison by training a 10-layer binary CNN model with/without dithering. Second, we construct test architectures for the CNN processing, and evaluate the hardware impact of adopting the dithering. Through the above, we show that the proposed algorithm can achieve higher accuracy than conventional binary quantization while minimizing the hardware cost.

5.6.1 Experimental Setup

Training Environment

To organize the neural network models, we used TensorFlow, Keras, and PyTorch frameworks running on GPU servers. We customized the frameworks to apply several non-standard operations. We implemented the forward and backward operations of the error diffusion using the TensorFlow and PyTorch C++/CUDA API according to the discussion in Section 5.5.1. The *Sign* activation function and weight binarization method proposed in [18] were also included in our implementation. In the process of the hyperparameter exploration, we partially used Optuna [65], though the configurations of the final experiments were manually fixed to equalize the environments along them.

The network models we tested are shown in Tables 5.1 and 5.2, namely Model A and Model B. Both in the "w/o Dither" and "w/ Dither" cases, the first convolutional layers accept the full-range (24-bit color) images and binary weights, and the later layers have the binarized activations and weights, except for the final readout layer that remains floating-point due to the Softmax operation. This would not be a limitation when the model is offloaded on an accelerator; most implementations of the neural network accelerators have the input layer preprocessed by their host CPUs, and the final readout Softmax layer is only applied in the training phase and is replaced with a linear or binary activation layer (this is possible because the Softmax operation is monotonic and only the location of the maximal value is important). The models were trained using the CIFAR-10 image recognition dataset [42] in Models A and B, and SVHN [61] dataset in Model B. To use

CIFAR-10 and SVHN whose input size is 32×32 in Model B with the input size 224×224, we enlarged the input images by bilinear interpolation. We used Adam optimizer with the learning rate decay strategy starting from 0.01 and halved every 25 epochs (Model A), and from 0.005 being halved every 20 epochs (Model B). We applied the data augmentation technique to improve the generalization capability of the networks.

The reason why we did not apply the dithering in DW-CONV layers in Model B is that they are followed by Conv layers with 1×1 kernels (*i.e.* pointwise convolution layers). The "demodulation" of the dithered signal in the context of signal processing is done by applying a low-pass filter; in a convolutional neural network, the convolutional kernels are expected to act as the filters. This suggests that the size of convolution kernels, as well as the number and location of layers with dithering, would be a key when we explore the hyperparameter space.

TABLE 5.1: TEST NETWORK MODEL ARCHITECTURES A

#	Layer Type [a]	Activation F. [b]		Output Size [c]
		w/o Dither	w/ Dither	
0	INPUT	-		$32 \times 32 \times 3$
1	Conv[d]	Sign	**Dither**	$30 \times 30 \times 128$
2	Conv	Sign	**Dither**	$30 \times 30 \times 128$
-	MaxPool	Sign	Sign	$15 \times 15 \times 128$
3	Conv	Sign		$15 \times 15 \times 256$
4	Conv	Sign		$15 \times 15 \times 256$
-	MaxPool	Sign		$7 \times 7 \times 256$
5	Conv	Sign		$7 \times 7 \times 512$
6	Conv	Sign		$7 \times 7 \times 512$
-	MaxPool	Sign		$3 \times 3 \times 512$
7	FC	Sign		1,024
8	FC	Sign		1,024
9	FC	SoftMax		10

[a] In *Layer Type*, we represent the input layer as INPUT, convolutional layers with 3×3 kernel as Conv, fully-connected layers as FC, max pooling layers with 2×2 window as MaxPool.

[b] In *Activation F.*, sign activation function is denoted as Sign and 1-directional dithering as Dither. A batch normalization layer is included in each Conv layer but is not shown in the table.

[c] *Output Size* refers to (Output height) $\times$ (Output width) $\times$ (Output channels) for the input and convolutional layers and (Output neurons) for the fully-connected layers.

[d] The first Convs #1 in both case are in the "valid" mode, and the others are in the "same" padding mode.

TABLE 5.2: TEST NETWORK MODEL ARCHITECTURES B: MOBILENETV1 [34]

#	Layer Type [a]	Activation F. [b]		Output Size [c]
		w/o Dither	w/ Dither	
0	INPUT	-		$224 \times 224 \times 3$
1	Conv 3-s2	Sign	**Dither**	$112 \times 112 \times 32$
2-1	DW-Conv 3-s1	Sign	Sign	$112 \times 112 \times 32$
2-2	Conv 1-s1	Sign	**Dither**	$112 \times 112 \times 64$
3-1	DW-Conv 3-s2	Sign	Sign	$56 \times 56 \times 64$
3-2	Conv 1-s1	Sign	**Dither**	$56 \times 56 \times 128$
4-1	DW-Conv 3-s1	Sign		$56 \times 56 \times 128$
4-2	Conv 1-s1	Sign		$56 \times 56 \times 128$
5-1	DW-Conv 3-s2	Sign		$28 \times 28 \times 128$
5-2	Conv 1-s1	Sign		$28 \times 28 \times 256$
6-1	DW-Conv 3-s1	Sign		$28 \times 28 \times 256$
6-2	Conv 1-s1	Sign		$28 \times 28 \times 256$
7-1	DW-Conv 3-s2	Sign		$14 \times 14 \times 256$
7-2	Conv 1-s1	Sign		$14 \times 14 \times 256$ [d]
8∼12-1	DW-Conv 3-s1	Sign		$14 \times 14 \times 512$
8∼12-2	Conv 1-s1	Sign		$14 \times 14 \times 512$
13-1	DW-Conv 3-s2	Sign		$7 \times 7 \times 512$
13-2	Conv 1-s1	Sign		$7 \times 7 \times 1{,}024$
14-1	DW-Conv 3-s1	Sign		$7 \times 7 \times 1{,}024$
14-2	Conv 1-s1	Sign		$7 \times 7 \times 1{,}024$
-	AvePool	-		$1 \times 1 \times 1{,}024$
15	Conv 1-s1	SoftMax		$1 \times 1 \times 10$

[a] In *Layer Type*, we represent the input layer as INPUT, standard and pointwise convolutional layers with $k \times k$ kernel and stride s as CONV k-ss, depthwise convolution as DW-CONV-k-ss, and an average pooling with 7×7 kernel and window (also known as global average pooling) layer as AvePool. All the CONV and DW-Conv layers are in "same" padding mode.

[b] In *Activation F.*, sign activation function is denoted as Sign, and 1-directional dithering as Dither. A batch normalization layer is included in each CONV or DW-Conv layer but is not shown in the table.

[c] *Output Size* refers to (Output height) × (Output width) × (Output channels).

[d] Although the output size of the layer #7-2 is $14 \times 14 \times 512$ in [34], we utilized $14 \times 14 \times 256$ due to the resource limitation of the training environment.

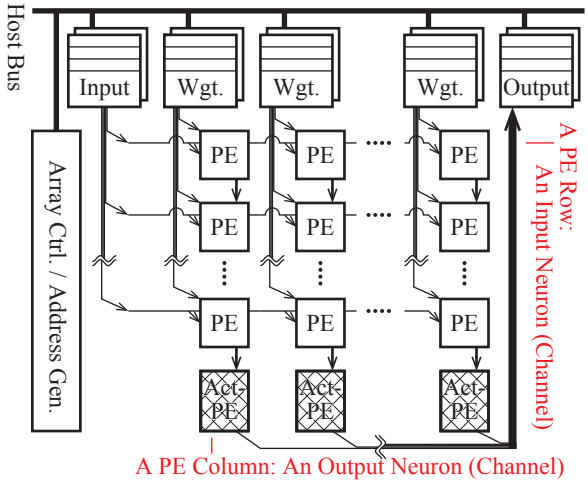

Fig. 5.6: Whole PE array with the input/weight/output RAMs and the controller in the prototype array architecture. The arithmetic bit width is denoted as b.

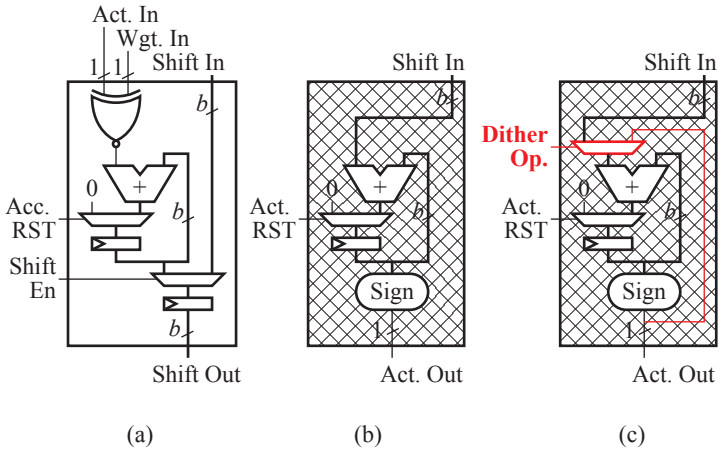

Fig. 5.7: PE structure. (a) A PE to calculate the weighted sum of an input channel, (b)(c) Act-PEs with/without dithering support to sum the results of PEs in a column and apply the Sign function to the sum. The arithmetic bit width is denoted as b.

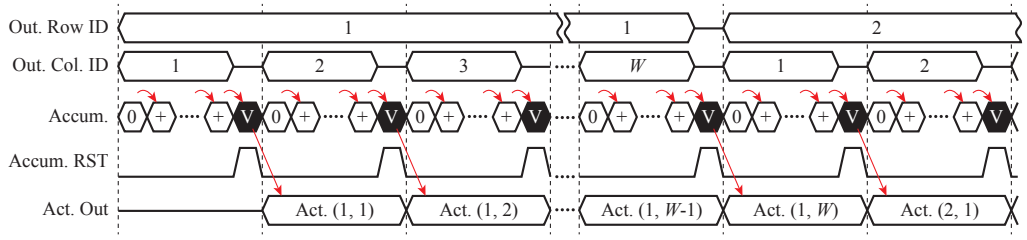

Fig. 5.8: Timing chart of the PE array without dithering. The channel axes are not shown. *W* denotes the output width, 'Accum.' is the value of the accumulator, and '+' and 'V' means 'accumulation' and 'valid' respectively.

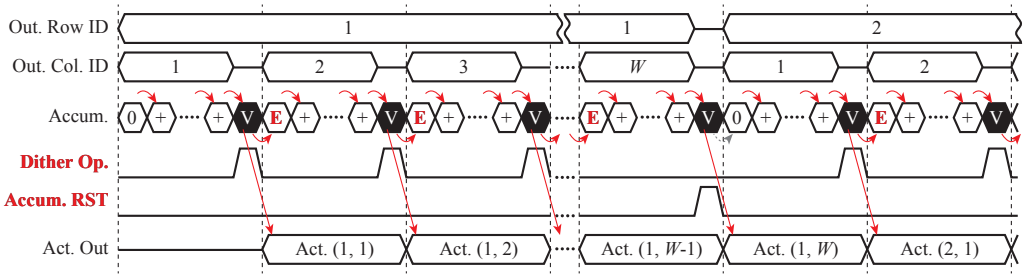

Fig. 5.9: Timing chart of the PE array with dithering. 'E' in 'Accum.' denotes the quantization error calculated using the previous 'V' and 'Act. Out' values.

Target FPGA

We designed the prototype architectures in Verilog HDL, and synthesized them using Xilinx Vivado 2017.4. We chose the Xilinx Zynq-7000 XC7Z020 FPGA mounted on the ZedBoard evaluation kit as the target because it is a middle-range system-on-chip (SoC) that features the ARM processor coupled with user logic on a single FPGA, which is an acceptable prototype candidate for mobile applications.

5.6.2 Accuracy

The accuracy evaluation was conducted on a GPGPU workstation using a 10-layer tiny CNN models "Model A" (Table 5.1) with CIFAR-10 dataset and a MobileNet[34]-based CNN models "Model B" using CIFAR-10 and SVHN format 2 dataset. We first pre-trained a binary neural network model with the same structure as the "w/o Dither" model for 200/100 epochs for Models A/B, and we then trained the two models "w/o Dither" and

TABLE 5.3: IMPLEMENTATION RESULT OF THE PROTOTYPE ARCHITECTURES

Resource		Avail.	w/o Dither		w/ Dither	
			Util.	←%	Util.	←%
LUTs		53,200	18,515	34.8	18,560	34.9
Registers		106,400	19,586	18.4	19,622	18.4
BRAMs	[Tiles]	140	72	51.4	72	51.4
	[kb]	5,040	2,592		2,592	
Block IOs		200	0	0.0	0	0.0
(Accuracy A [%])		CIFAR-10		85.83		87.14
(Accuracy B [%])		CIFAR-10		73.45		75.15
		SVHN		90.91		91.64

* For both architectures, the bit width of the accumulators was set to 12; the numbers of PE rows and columns were set to 16; and all the RAMs were 16-bit 4k-word dual-bank.

"w/ Dither" for 200/50 more epochs starting with the pre-trained weights.

The Model A "w/ Dither" achieved 87.14% accuracy with two layers employing the dithering, whereas the baseline "w/o Dither" model was 85.83% accurate. This result outperformed the previous multithreshold model [26], although a straightforward comparison is impossible due to the differences in the model structure. It should be noted that the proposed method could be used with other state-of-the-art algorithms including [26].

Model B trained with CIFAR-10 dataset showed the accuracy 75.15%/73.45% with/without dithering, while the model trained using SVHN format 2 achieved 91.64%/90.91% with/without dithering, respectively. The improvement with SVHN dataset is not greater than that with CIFAR-10 dataset, although the baseline accuracy with SVHN is higher than CIFAR-10. One possible explanation of this is the following: dithering remaps the multi-level bit precision on the spatial resolution using multiple low-precision pixels, which assumes the large enough area of smooth gradation; the pictures of digits in the SVHN dataset did not match this assumption.

Again, the error-diffusion-based dithering algorithm can be utilized with any settings of the dataset, network model structure, and optimizer, as long as the model has 2-d (height, width) or higher-dimensional image-like activations with quantization, however it works well especially with spatially smooth data like objects in natural photos.

5.6.3 FPGA Implementation

To evaluate the hardware impact on the dithering operation, we implemented two prototype PE-array-based parallel architectures on an FPGA with/without dithering support. This design only supports the processing of a convolutional layer of a neural network for simplicity.

Figs. 5.6 and 5.7(a)(b)(c) indicate the prototype architecture. The PE array (Fig. 5.6 forms a primitive binary neural network accelerator based on the typical output-input-channel parallelism, where each PE row corresponds to an input channel and each PE column corresponds to an output channel. This is a binary-only subset of a single core of the architecture proposed in [79]; the weights and inputs/outputs are all in 1-bit. In this configuration, an input activation is shared among multiple output channels (*i.e.* among PEs in a row), and an output is calculated in a column, with a word of the weight RAM being distributed bit-wise to the PEs and the Activation PE (Act-PE; Fig. 5.7(b)) summing up the partial sums (weighted-sums) accumulated in each PE (shifted column-wise sequentially) and generating an output activation.

The only difference between the two architectures with/without dithering is the type of Act-PE (without dithering Fig. 5.7(b) or with dithering (c)) used. The dithering Act-PE in Fig. 5.7(c) has additional multiplexers to select either RESET or DITHER operation at the end of the input accumulation. Repeatedly, the dithering (error diffusion) operation can be performed as a subtraction (accumulation), for which the adder in an accumulator of each Act-PE can be used. Therefore, the overhead of the hardware resource for adopting the dithering operation would not be significant. In addition, since the dithering operation is conducted *instead of* the accumulator reset operation, no additional clock cycles are needed, as shown in Figs. 5.8 and 5.9.

Here, the implementation results of the architectures are shown in Table 5.3. The table includes only the PE array and its corresponding controllers and RAMs; any other parts such as data transfer circuits are not included. In this evaluation, we used the Xilinx Vivado software and the Zynq-7000 XC7Z020 FPGA, as mentioned above, with both architectures synthesized/PARed by the "Area Exploration" strategy. As seen in the table, the LUT and register usage would increase by less than 1% upon adding the dithering operation, and the RAM usage does not change, because the dithering operation does not need any additional arithmetic units. Therefore, the proposed dithering algorithm is proved to be a hardware-friendly technique that can be utilized with a very few additional hardware resources.

5.7 Other Quantization Techniques with Dithering

We have discussed and evaluated the dithering mainly in the binary neural network and its hardware as an example. The algorithm would be extended to any kind of quantization methods, such as ternary [49], logarithmic [53], and linear fixed-point, since any quantization technique produces the quantization error.

Here, we discuss applying the error diffusion to a typical hardware architecture for neural networks with fixed-point and logarithmic quantization. Quite similar methodology to the binary accelerator with dithering we used for evaluation could be utilized here, therefore the hardware overhead would not be significant even for that quantized hardware, although they have not been synthesized and evaluated.

5.7.1 Fixed-Point Linear Quantization

Fixed-point linear quantization has been utilized in various neural network accelerator arcitectures. In the fixed-point quantization with the B_i-bit integer part and B_f-bit fractional part (*i.e.* QB_i.B_f format), the multiplication of two numbers produces the product with the $2B_i$ integer and $2B_f$ fractional bits, and they are then truncated to the original B_i-and-B_f-bit expression, as shown in Fig. 5.10. Here, the remainder bits of the integer part corresponds to the arithmetic overflow, but that of the fractional part is simply ignored, thus this causes the *quantization error*.

In a real fixed-point MAC operation, the bit truncation usually takes place after the completion of the accumulation, not just after the multiplication, therefore the accumulator has the bit width for at least $2B_i$-and-$2B_f$-bit number. An activation function, such as ReLU, is applied to the produced sum. Thus the same thing as the binary neural network — the quantization error occurring at the end of MAC computation appears also in the fixed-point quantization. The computation of the dithering on the fixed-point number is quite similar to that of binary, but the biggest difference from the binary is that the resultant number is still in a multi-level number. As indicated in Fig. 5.11, the difference between the accumulated sum and the truncated result is added at the accumulation of the next output neuron as the *quantization error*. Similarly to the case of binary, the method where the quantization error is accumulated in the accumulator instead of the reset operation can be used here. If the bit truncation is implemented as *rounding toward zero*, the quantization error can be computed by picking the lower fractional bits (masking the larger bits and extending its sign bit) with no explicit subtraction as shown in Fig. 5.12.

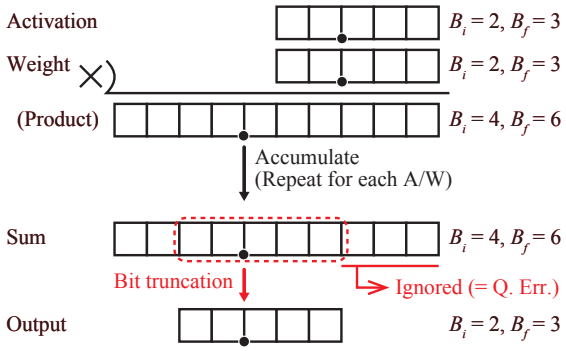

Fig. 5.10: Typical procedure of the MAC operation of the fixed-point activations and weights.

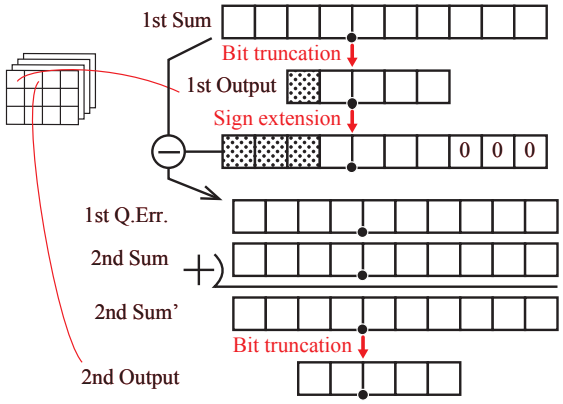

Fig. 5.11: Dithering operation in the fixed-point computation.

5.7.2 Other Quantization Methods

Neural network using logarithmic quantization was proposed in [53], where both the activation and weights are represented in base-2 logarithm. The advantage of the logarithmic quantization is that the multipliers can be eliminated, because the multiplication between the weight and activation is replaced by the addition between the logarithms of them.

Though [53] presented a technique to compute all the operation including arithmetic addition in the logarithmic domain, one possible and reasonable realization of the logarithmic quantization is to represent/multiply the activation and weight in the logarithmic domain and to accumulate the product in the linear domain. As shown in Fig. 5.13, after

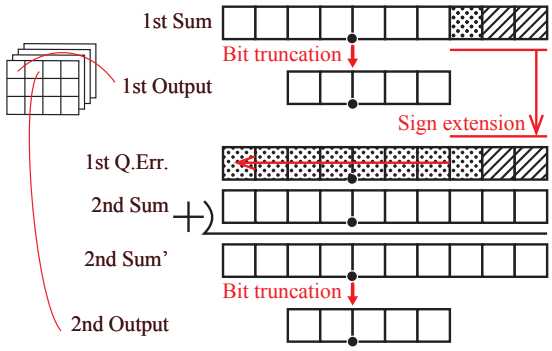

Fig. 5.12: Dithering operation in the fixed-point computation when the truncation is rounding toward zero.

the multiplication (addition between the logalithmic-represented activation and weight), the product is converted into the linear expression with proper scaling, and then it is accumulated in the linear domain, and finally the sum is activated and quantized in logarithmic expression as the output activation at the end of accumulation. Therefore, the *quantization error* can be defined as the difference between the linearly accumulated sum and the real value of the logarithmic-represented activation.

Fig. 5.14 indicates the ideal and primitive architecture to process dithering in logarithmic neural networks. Since the accumulation of the products of the activations and weights is conducted in linear domain, the same processing method as the case of fixed-point can be used; the quantization error to be accumulated is computed as the subtraction between converted logarithmic-expressed activation and the accumulator value using the adder of the accumulator.

5.8 Conclusion

We have proposed the dithering neural network to improve the accuracy of the quantized neural network models. The experimental accelerator architectures using binary neural networks have proved that the proposed concept can be efficiently realized without deviating from the nature of the quantized neural network on the limited hardware resources.

This method was motivated by the fact the processing on a MAC-array-based neural network processor can be interpreted from the signal processing viewpoint. It was the

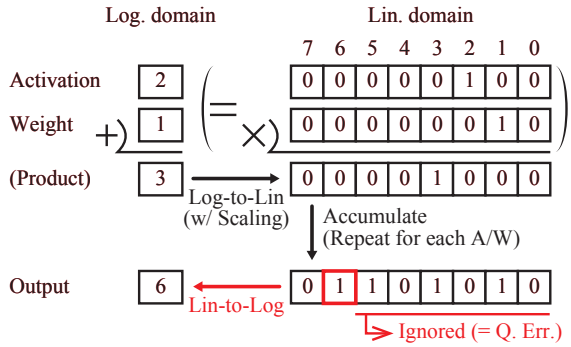

Fig. 5.13: Typical procedure of the MAC operation of the logarithmic activations and weights. In this example, the scaling factor is omitted for the simplicity, and the sign is coded in the *signed absolute* expression.

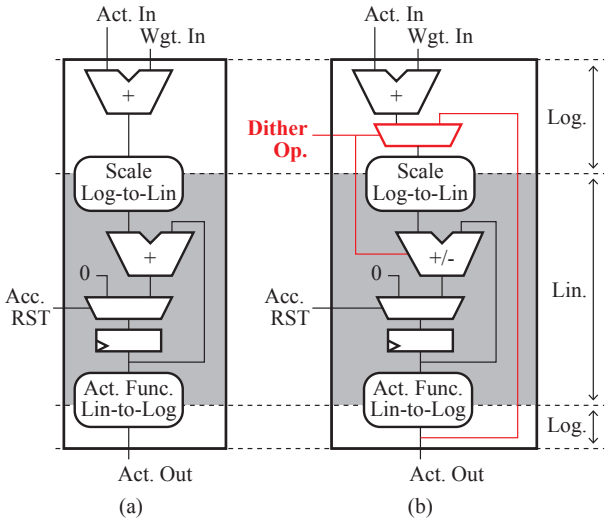

Fig. 5.14: (a) A typical logarithmic PE (for output-pixel-parallel computation). (b) Applying the dithering in the logarithmic PE. These PEs are a straightforward extension from the binary ones in Fig. 5.3.

first contribution that enhances the quantized neural network with a very few additional hardware resource requirements by importing the quantization error minimizing technique from the field of image processing as the fruit of the hardware-algorithm co-designing strategy.

Chapter 6

Conclusion

Deep neural networks opened the door for AI-powered smart society. Since the dawn of this era of AI, the advance in hardware technology and algorithm has driven the explosive evolution of information processing.

The algorithms are going through a fast transition reflecting the world's application demands, while the limits of straightforward and strong-arm development are approaching in model structure and hardware scaling. In this sense, the significance of hardware-software coevolution is getting higher. This thesis presented an example of a methodology for constructing highly efficient edge-side neural network processors by the comprehensive approach, proving the effectiveness of interdisciplinary hardware-software co-optimization.

In Chapter 3, we designed an output-stationary parallel computation array architecture that can process convolutional (Conv) and fully-connected (FC) layers on a single array efficiently. Despite the differences in computational procedure and data topology, we quarried the one-to-all multiply-addition (MAC) operation as the atomic computation primitive. Conv processing can be done by picking a weight value from the kernel and conducting MAC with its corresponding input pixels, and an FC layer can be computed by parallel MAC between an input neuron and multiple weight values. According to this thought, we first designed an ideal and simplified data flow architecture using the shared and individual buses on the processing element (PE) array. To construct this into a practical one, we had to relax the internal data rate and memory addressing complexity. The row-wise forwarding buses, multi-bank memory architecture, and multithreaded accumulators accomplished it. We confirmed the algorithmic requirements of Conv and FC layer configuration for achieving higher efficiency through the performance evaluation of this architecture.

Chapter 4 proposed a near-memory neural network processor named BRein Memory. The energy and latency of external data access have been significant concerns in low-power embedded systems. If all the parameters are stored in on-chip SRAMs, and the

computation is completed nearby the SRAMs without intermediate data load and store, the external memory can be omitted. In this sense, we considered near-memory processing units with a binary neural network algorithm. It restricts all the activations and weight parameters to be ± 1, which can be represented in one bit. The binary activation and weight occupy much less memory compared to fixed-point or standard floating-point expression. In addition, the multiplication between the binary numbers can be calculated as a single bitwise XNOR operation, for which the computational resource demands are drastically reduced. Thanks to these natures, we constructed a thin parallel computation unit between two SRAM macros, with the SRAM data pattern harmonizing with the neural network layers' potential parallelism. This structure encloses the parallel computation and data movements only near the SRAMs; the data movements between the computation units are serial, making it easy to stack computation units to process deeper neural networks without energy-consuming parallel buses. We proved the concept of highly efficient neural network processor architecture featuring near-memory processing through the fabricated LSI evaluation.

Chapter 5 discussed a hardware-oriented quantized neural network algorithm named Dither NN. Binary neural networks gain efficiency and performance, but there is an accuracy drop observed. Thus, it comes with a trade-off between accuracy and efficiency. When we see a neural network processor from the viewpoint of signal processing, we can find that the computation conducted in it can be interpreted as signal processing. Inspired by the dithering algorithm used for low-bit-width audio and image quantization, we proposed using dithering in quantized neural network hardware. Mapping the data dependency of dithering among temporally continual processing phases and using the error diffusion algorithm that can be computed by simple subtraction, this procedure does not incur inflation of spatial and temporal resources. We successfully integrated this algorithm into a prototyped parallel neural network accelerator on an FPGA. The evaluation indicated that dithering could improve the accuracy of binary neural networks with very few additional hardware resources. Reinterpreting the information representation and its processing brought accuracy improvement without degrading the efficiency.

The discussion of data delivery on a parallel computation array in Chapter 3 showed the significance of the data movements to the efficiency (especially in an FC layer processing). This led us to the idea of near-memory processing, proposed in Chapter 4, to eliminate the expensive parallel data transfer. The quantization techniques were the enabler of the high-efficiency near-memory neural network processing, reducing computational and memory resource demands, but the accuracy degradation became a new problem. Chapter 5 addressed this issue by importing the idea of low-bit-width quantization with dithering from

the field of image processing into the neural network processor. Dithering quantization using the error diffusion algorithm improved the accuracy with negligible hardware overhead by reflecting the computational procedures and structures of the quantized neural network architecture.

The contributions of this thesis include the following:

- Dataflow-centric programmable hardware enabled by observing the potentiality of parallel MACs universal among the target tasks, as well as the practical example of architecture exploration and optimization focusing on data access patterns.
- Near-memory neural network processing that empowers massively parallelized and highly efficient computation with quantization techniques.
- The idea of hardware-oriented quantization inspired by signal processing, along with the neural network viewpoint for obtaining its fundamental nonlinearity and backpropagation-capable approximation.

We have discussed and proved these concepts through FPGA and ASIC prototyping, showing an example of architectural optimization and algorithm improvements by gazing at the bases of computation required in them. The key to highly efficient programmable processors is to distill universal and atomic patterns from the underlying essence of computational structure and data topology while following the trends of ever-changing neural network algorithms and hardware. We practiced this philosophy through exploring a data-flow-reconfigurable architecture, constructing a compact near-memory processor utilizing quantization techniques, and conversely reflecting the hardware structure into a quantization algorithm.

The neural network algorithms continue to evolve. There should come new ideas, attempts, and applications one after another, as the market and needs are expanding while the principle of the neural network explaining its ability has not been discovered completely. For instance, a new hypothesis that a successful neural network model is a superposition of appropriate subsets of the weight and connections has been studied recently [24, 67], which could change the neural network model exploration and deployment schemes. Even if the neural network application and situation experienced transition, the essential concepts and methodology discussed above for constructing an efficient inference platform would remain effective and meaningful. The hardware technology and processor architectures have a great responsibility for promoting those advances in the model algorithms, and also have a chance of evolution enjoying these innovations.

Acknowledgements

I would like to express the deepest appreciation to my supervisor, Professor Masato Motomura. He has supervised me for seven years. It is my honor to start my academic career under the guidance of him. He has also supported and encouraged me in my daily life. I would also thank his wife, Mrs. Naomi Motomura. She supported me when I became independent of my parents to move to Kanagawa.

I also would like to express my gratitude to Professor Tetsuya Asai for guiding me for many years. My decision to advance to the doctoral program was owing to his encouragement at VLSI Symposium in Kyoto.

I would thank my sub-advisors, Professor Jaehoon Yu, Professor Shinya Takamaeda-Yamazaki, Professor Thiem Van Chu, and Professor Kazushi Kawamura, for ceaseless advice throughout the research and life in the laboratory.

I sincerely appreciate the examiners, Professor Atsushi Takahashi, Professor Hiroshi Sasaki, Professor Hiroki Nakahara, Professor Yuko Hara-Azumi, and Professor Tsuyoshi Isshiki for their constructive criticism and advice on the preparation of the thesis, as well as on the preexamination.

I am grateful to my coauthors and collaborators, Professor Masayuki Ikebe, Dr. Hiroshi Momose in Hokkaido University, Professor Tadahiro Kuroda in Keio University, Dr. Takao Toi, Dr. Taro Fujii in Renesas Electronics, Dr. Kodai Ueyoshi in KU Leuven, Mr. Kazutoshi Hirose, Mr. Haruyoshi Yonekawa, Mr. Akira Jinguji, Mr. Naoto Soga in Tokyo Institute of Technology, and Ms. Yuka Oba in Hokkaido University, for their attentive cooperation on the paper writing, experiment setup, and technical discussion.

I would thank my friends and lab mates, Mr. Naoto Iwamaru, Mr. Kentaro Orimo, Ms. Aoi Tanibata, Mr. Takayuki Yoshida, Mr. Seokjin Na, Dr. Kasho Yamamoto, Mr. Ryota Uematsu, Mr. Takeshi Shimada, Mr. Takumi Kudo, Mr. Tatsuya Kaneko, Mr. Taiga Ikeda, Mr. Yuki Hirayama, Mr. Kaisei Okawa, Mr. Shungo Kumazawa, Mr. Junnosuke Suzuki, Mr. Yoshiharu Yamagishi, Mr. Yasuyuki Okoshi, Mr. Tomohiro Kaneko, Mr. Ryuichi Kitajima, Mr. Iriya Takayama, Dr. Eric-Shun Fukuda, Dr. Kazuyoshi Ishimura, Dr. Itaru Hida, and all current and past colleagues in AI Computing Research Unit (ArtIC), Tokyo Institute of Technology, and Laboratories for Advanced LSI Engineering

(LALSIE), Hokkaido University.

I would like to thank the secretaries Ms. Juri Hashimoto, Ms. Taeko Tsuchiya in Tokyo Institute of Technology, Ms. Yuki Miura, and Ms. Atsuko Yokokawa in Hokkaido University. My research in the lab has been performed with their help.

This work was partially supported by JST ACCEL Grant Number JPMJAC1502, JSPS KAKENHI Grant Numbers JP18J20307, JP18H05288, and JST PRESTO JPMJPR18M9.

I transferred from Hokkaido University to Tokyo Institute of Technology in my second year of the Ph.D. program, which has broadened my horizons and gained my confidence.

Lastly, I express my gratitude to my parents and grandparents for their devotion.

Bibliography

[1] Hp labs : Cacti. `http://www.hpl.hp.com/research/cacti/` (web interface `http://quid.hpl.hp.com:9081/cacti/sram.y` is no longer accessible as of Jan. 19, 2021).

[2] K. Ando, K. Orimo, K. Ueyoshi, M. Ikebe, T. Asai, and M. Motomura. Reconfigurable processor array architecture for deep convolutional neural cetworks. In *2016 The 20th Workshop on Synthesis And System Integration of Mixed Information Technologies (SASIMI)*, Oct 2016.

[3] K. Ando, K. Orimo, K. Ueyoshi, H. Yonekawa, S. Sato, H. Nakahara, M. Ikebe, T. Asai, S. Takamaeda-Yamazaki, T. Kuroda, and M. Motomura. BRein Memory: A 13-layer 4.2 K neuron/0.8 M synapse binary/ternary reconfigurable in-memory deep neural network accelerator in 65 nm CMOS. In *2017 IEEE Symposium on VLSI Circuits (VLSI-Circuits)*, pages C24–C25, Kyoto, Japan, 2017.

[4] K. Ando, S. Takamaeda-Y., M. Ikebe, T. Asai, and M. Motomura. A multithreaded CGRA for convolutional neural network processing. *Circuits and Systems*, 8(6):149–170, June 2017. `doi:doi:10.4236/cs.2017.86010`.

[5] K. Ando, K. Ueyoshi, K. Hirose, K. Orimo, H. Yonekawa, S. Sato, H. Nakahara, M. Ikebe, S. Takamaeda-Yamazaki, T. Asai, and M. Motomura. In-memory area-efficient signal streaming processor design for binary neural networks. In *60th IEEE International Midwest Symposium on Circuits and Systems (MWSCAS)*, Boston, MA, USA, 2017.

[6] K. Ando, K. Ueyoshi, Y. Oba, K. Hirose, R. Uematsu, T. Kudo, M. Ikebe, T. Asai, S. Takamaeda-Yamazaki, and M. Motomura. Dither nn: An accurate neural network with dithering for low bit-precision hardware. In *2018 International Conference on Field-Programmable Technology (FPT)*, pages 6–13, 2018. `doi:10.1109/FPT.2018.00013`.

[7] K. Ando, K. Ueyoshi, Y. Oba, K. Hirose, R. Uematsu, T. Kudo, M. Ikebe, T. Asai, S. Takamaeda-Yamazaki, and M. Motomura. Dither nn: Hardware/algorithm co-design for accurate quantized neural networks. *IEICE Transactions on Information and Systems*, E102.D(12):2341–2353, 2019. doi:10.1587/transinf.2019PAP0009.

[8] K. Ando, K. Ueyoshi, K. Orimo, H. Yonekawa, S. Sato, H. Nakahara, S. Takamaeda-Yamazaki, M. Ikebe, T. Asai, T. Kuroda, and M. Motomura. BRein Memory: A single-chip binary/ternary reconfigurable in-memory deep neural network accelerator achieving 1.4 TOPS at 0.6 W. *IEEE Journal of Solid-State Circuits*, 53(4):983–994, April 2018. doi:10.1109/JSSC.2017.2778702.

[9] N. Burgess, J. Milanovic, N. Stephens, K. Monachopoulos, and D. Mansell. Bfloat16 processing for neural networks. In *2019 IEEE 26th Symposium on Computer Arithmetic (ARITH)*, pages 88–91, 2019. doi:10.1109/ARITH.2019.00022.

[10] H. Cai, L. Zhu, and S. Han. ProxylessNAS: Direct neural architecture search on target task and hardware. *CoRR*, abs/1812.00332, 2018. URL: http://arxiv.org/abs/1812.00332, arXiv:1812.00332.

[11] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, page 269–284, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2541940.2541967.

[12] Y. Chen, J. Emer, and V. Sze. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 367–379, 2016. doi:10.1109/ISCA.2016.40.

[13] Y. 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*, 52(1):127–138, 2017. doi:10.1109/JSSC.2016.2616357.

[14] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam. Dadiannao: A machine-learning supercomputer. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 609–622, Dec 2014. doi:10.1109/MICRO.2014.58.

[15] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam. Dadiannao: A machine-learning supercomputer. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, pages 609–622, Washington, DC, USA, 2014. IEEE Computer Society. URL: http://dx.doi.org/10.1109/MICRO.2014.58, doi:10.1109/MICRO.2014.58.

[16] Y. Chen, T. 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*, 9(2):292–308, 2019. doi:10.1109/JETCAS.2019.2910232.

[17] Y. H. Chen, T. Krishna, J. Emer, and V. Sze. 14.5 Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. In *2016 IEEE International Solid-State Circuits Conference (ISSCC)*, pages 262–263, Jan 2016. doi:10.1109/ISSCC.2016.7418007.

[18] M. Courbariaux and Y. Bengio. BinaryNet: Training deep neural networks with weights and activations constrained to +1 or -1. *CoRR*, abs/1602.02830:1–11, 2016. URL: http://arxiv.org/abs/1602.02830.

[19] M. Courbariaux, Y. Bengio, and J. David. Low precision arithmetic for deep learning. *CoRR*, abs/1412.7024, 2014. URL: http://arxiv.org/abs/1412.7024.

[20] M. Courbariaux, Y. Bengio, and J. David. BinaryConnect: Training deep neural networks with binary weights during propagations. *CoRR*, abs/1511.00363, 2015. URL: http://arxiv.org/abs/1511.00363.

[21] M. Courbariaux, Y. Bengio, and J. David. Training deep neural networks with low precision multiplications. *arXiv*, 1412.7024v5, 2015. URL: http://arxiv.org/abs/1412.7024v5, arXiv:1412.7024v5.

[22] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio. Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1. *ArXiv e-prints*, Feb. 2016. arXiv:1602.02830.

[23] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam. Shidiannao: Shifting vision processing closer to the sensor. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 92–104, 2015. doi:10.1145/2749469.2750389.

[24] J. Frankle and M. Carbin. The lottery ticket hypothesis: Training pruned neural networks. *CoRR*, abs/1803.03635, 2018. URL: http://arxiv.org/abs/1803.03635, arXiv:1803.03635.

[25] R. Fuchikami and F. Issiki. Fast and light-weight binarized neural network implemented in an fpga using lut-based signal processing and its time-domain extension for multi-bit processing. In *2019 IEEE 9th International Conference on Consumer Electronics (ICCE-Berlin)*, pages 120–121, 2019. doi:10.1109/ICCE-Berlin47944.2019.8966187.

[26] M. Ghasemzadeh, M. Samragh, and F. Koushanfar. ReBNet: Residual binarized neural network. *CoRR*, abs/1711.01243, 2017. URL: http://arxiv.org/abs/1711.01243, arXiv:1711.01243.

[27] K. Guo, S. Zeng, J. Yu, Y. Wang, and H. Yang. [dl] a survey of fpga-based neural network inference accelerators. *ACM Trans. Reconfigurable Technol. Syst.*, 12(1), Mar. 2019. doi:10.1145/3289185.

[28] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan. Deep learning with limited numerical precision. In *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*, ICML'15, pages 1737–1746. JMLR.org, 2015. URL: http://dl.acm.org/citation.cfm?id=3045118.3045303.

[29] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally. Eie: Efficient inference engine on compressed deep neural network. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, page 243–254. IEEE Press, 2016. doi:10.1109/ISCA.2016.30.

[30] S. Han, H. Mao, and W. J. Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv*, arXiv:1510.00149, 2015. URL: http://arxiv.org/abs/1510.00149, arXiv:1510.00149.

[31] S. Han, J. Pool, J. Tran, and W. J. Dally. Learning both weights and connections for efficient neural networks. *CoRR*, abs/1506.02626, 2015. URL: http://arxiv.org/abs/1506.02626, arXiv:1506.02626.

[32] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

[33] A. Howard, M. Sandler, G. Chu, L.-C. Chen, B. Chen, M. Tan, W. Wang, Y. Zhu, R. Pang, V. Vasudevan, Q. V. Le, and H. Adam. Searching for mobilenetv3. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, October 2019.

[34] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam. MobileNets: Efficient convolutional neural networks for mobile vision applications. *CoRR*, abs/1704.04861, 2017. URL: http://arxiv.org/abs/1704.04861, arXiv:1704.04861.

[35] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio. Quantized neural networks: Training neural networks with low precision weights and activations. *CoRR*, abs/1609.07061, 2016. URL: http://arxiv.org/abs/1609.07061.

[36] Intel. BFLOAT16 – hardware numerics definition. URL: https://software.intel.com/sites/default/files/managed/40/8b/bf16-hardware-numerics-definition-white-paper.pdf, accessed Jan 19, 2021, published Nov. 2018.

[37] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015. URL: http://arxiv.org/abs/1502.03167.

[38] Jian Ouyang, Shiding Lin, Wei Qi, Yong Wang, Bo Yu, and Song Jiang. Sda: Software-defined accelerator for large-scale dnn systems. In *2014 IEEE Hot Chips 26 Symposium (HCS)*, pages 1–23, 2014. doi:10.1109/HOTCHIPS.2014.7478821.

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

E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, page 1–12, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3079856.3080246.

[40] D. D. Kalamkar, D. Mudigere, N. Mellempudi, D. Das, K. Banerjee, S. Avancha, D. T. Vooturi, N. Jammalamadaka, J. Huang, H. Yuen, J. Yang, J. Park, A. Heinecke, E. Georganas, S. Srinivasan, A. Kundu, M. Smelyanskiy, B. Kaul, and P. Dubey. A study of BFLOAT16 for deep learning training. *CoRR*, abs/1905.12322, 2019. URL: http://arxiv.org/abs/1905.12322, arXiv:1905.12322.

[41] S. Kang, D. Han, J. Lee, D. Im, S. Kim, S. Kim, and H. Yoo. 7.4 GANPU: A 135tflops/w multi-DNN training processor for GANs with speculative dual-sparsity exploitation. In *2020 IEEE International Solid- State Circuits Conference - (ISSCC)*, pages 140–142, 2020. doi:10.1109/ISSCC19947.2020.9062989.

[42] A. Krizhevsky and G. Hinton. Learning multiple layers of features from tiny images. *Computer Science Department, University of Toronto*, Jan 2009.

[43] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, pages 1097–1105, 2012.

[44] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998. doi:10.1109/5.726791.

[45] E. H. Lee, D. Miyashita, E. Chai, B. Murmann, and S. S. Wong. Lognet: Energy-efficient neural networks using logarithmic computation. In *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5900–5904, 2017. doi:10.1109/ICASSP.2017.7953288.

[46] J. Lee, C. Kim, S. Kang, D. Shin, S. Kim, and H. Yoo. Unpu: An energy-efficient deep neural network accelerator with fully variable weight bit precision. *IEEE Journal of Solid-State Circuits*, 54(1):173–185, 2019. doi:10.1109/JSSC.2018.2865489.

[47] J. Lee, C. Kim, S. Kang, D. Shin, S. Kim, and H. J. Yoo. UNPU: A 50.6TOPS/W unified deep neural network accelerator with 1b-to-16b fully-variable weight bit-precision. In *2018 IEEE International Solid - State Circuits Conference - (ISSCC)*, pages 218–220, Feb 2018. doi:10.1109/ISSCC.2018.8310262.

[48] J. Lee, J. Lee, D. Han, J. Lee, G. Park, and H. Yoo. 7.7 lnpu: A 25.3tflops/w sparse deep-neural-network learning processor with fine-grained mixed precision of fp8-fp16. In *2019 IEEE International Solid- State Circuits Conference - (ISSCC)*, pages 142–144, 2019. doi:10.1109/ISSCC.2019.8662302.

[49] F. Li and B. Liu. Ternary weight networks. *CoRR*, abs/1605.04711:1–5, 2016. URL: http://arxiv.org/abs/1605.04711.

[50] W. Lin, D. Tsai, L. Tang, C. Hsieh, C. Chou, P. Chang, and L. Hsu. Onnc: A compilation framework connecting onnx to proprietary deep learning accelerators. In *2019 IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS)*, pages 214–218, 2019. doi:10.1109/AICAS.2019.8771510.

[51] P. Merolla, J. Arthur, F. Akopyan, N. Imam, R. Manohar, and D. S. Modha. A digital neurosynaptic core using embedded crossbar memory with 45pj per spike in 45nm. In *2011 IEEE Custom Integrated Circuits Conference (CICC)*, pages 1–4, Sept 2011. doi:10.1109/CICC.2011.6055294.

[52] P. A. Merolla, J. V. Arthur, R. Alvarez-Icaza, A. S. Cassidy, J. Sawada, F. Akopyan, B. L. Jackson, N. Imam, C. Guo, Y. Nakamura, B. Brezzo, I. Vo, S. K. Esser, R. Appuswamy, B. Taba, A. Amir, M. D. Flickner, W. P. Risk, R. Manohar, and D. S. Modha. A million spiking-neuron integrated circuit with a scalable communication network and interface. *Science*, 345(6197):668–673, 2014. URL: http://science.sciencemag.org/content/345/6197/668, arXiv:http://science.sciencemag.org/content/345/6197/668.full.pdf, doi:10.1126/science.1254642.

[53] D. Miyashita, E. H. Lee, and B. Murmann. Convolutional neural networks using logarithmic data representation. *CoRR*, abs/1603.01025:1–10, 2016. URL: http://arxiv.org/abs/1603.01025, arXiv:1603.01025.

[54] B. Moons, B. D. Brabandere, L. V. Gool, and M. Verhelst. Energy-efficient convnets through approximate computing. In *2016 IEEE Winter Conference on Applications of Computer Vision (WACV)*, pages 1–8, March 2016. doi:10.1109/WACV.2016.7477614.

[55] B. Moons, R. Uytterhoeven, W. Dehaene, and M. Verhelst. 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)*, pages 246–247, Feb 2017. doi:10.1109/ISSCC.2017.7870353.

[56] B. Moons and M. Verhelst. A 0.3 –2.6 TOPS/W precision-scalable processor for real-time large-scale convnets. In *2016 IEEE Symposium on VLSI Circuits (VLSI-Circuits)*, pages 1–2, June 2016. doi:10.1109/VLSIC.2016.7573525.

[57] B. Moons and M. Verhelst. A 0.3-2.6 TOPS/W precision-scalable processor for real-time large-scale convnets. In *2016 Symposium on VLSI Circuits Digest of Technical Papers (VLSI)*, pages 178–179, Jun 2016.

[58] H. Nakahara, Y. Sada, M. Shimoda, K. Sayama, A. Jinguji, and S. Sato. Fpga-based training accelerator utilizing sparseness of convolutional neural network. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, pages 180–186, 2019. doi:10.1109/FPL.2019.00036.

[59] H. NAKAHARA, H. YONEKAWA, T. FUJII, M. SHIMODA, and S. SATO. Guinness: A gui based binarized deep neural network framework for software programmers. *IEICE Transactions on Information and Systems*, E102.D(5):1003–1011, 2019. doi:10.1587/transinf.2018RCP0002.

[60] H. Nakahara, H. Yonekawa, T. Sasao, H. Iwamoto, and M. Motomura. A memory-based realization of a binarized deep convolutional neural network. In *2016 International Conference on Field-Programmable Technology (FPT)*, pages 277–280, Dec 2016. doi:10.1109/FPT.2016.7929552.

[61] Y. Netzer, T. Wang, A. Coates, A. Bissacco, B. Wu, and A. Y Ng. Reading digits in natural images with unsupervised feature learning. *NIPS Workshop on Deep Learning and Unsupervised Feature Learning*, 01 2011. URL: http://ufldl.stanford.edu/housenumbers.

[62] H. Noguchi, K. Nomura, K. Abe, S. Fujita, E. Arima, K. Kim, T. Nakada, S. Miwa, and H. Nakamura. D-mram cache: Enhancing energy efficiency with 3t-1mtj dram / mram hybrid memory. In *2013 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1813–1818, March 2013. doi:10.7873/DATE.2013.363.

[63] E. Nurvitadhi, D. Sheffield, J. Sim, A. Mishra, G. Venkatesh, and D. Marr. Accelerating binarized neural networks: Comparison of fpga, cpu, gpu, and asic. In *2016 International Conference on Field-Programmable Technology (FPT)*, pages 77–84, Dec 2016. doi:10.1109/FPT.2016.7929192.

[64] NVIDIA. Nvidia Tesla V100 GPU architecture: The world's most advanced data center GPU. URL: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf, accessed Jan 19, 2021.

[65] Preferred Networks, Inc. Optuna: Define-by-run hyperparameter optimization framework. https://optuna.org/, 2018.

[66] A. Radford, L. Metz, and S. Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*, 2015.

[67] V. Ramanujan, M. Wortsman, A. Kembhavi, A. Farhadi, and M. Rastegari. What's hidden in a randomly weighted neural network? In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2020.

[68] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi. XNOR-Net: Imagenet classification using binary convolutional neural networks. *CoRR*, abs/1603.05279, 2016. URL: http://arxiv.org/abs/1603.05279.

[69] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

[70] F. Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[71] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

[72] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen. MobileNetV2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.

[73] D. Shin, J. Lee, J. Lee, and H. J. Yoo. 14.2 DNPU: An 8.1TOPS/W reconfigurable cnn-rnn processor for general-purpose deep neural networks. In *2017 IEEE International Solid-State Circuits Conference (ISSCC)*, pages 240–241, Feb 2017. doi:10.1109/ISSCC.2017.7870350.

[74] L. Sifre and S. Mallat. Rigid-motion scattering for image classification. *Ph. D. thesis*, 2014.

[75] J. Sim, J.-S. Park, M. Kim, D. Bae, Y. Choi, and L.-S. Kim. 14.6 a 1.42TOPS/W deep convolutional neural network recognition processor for intelligent ioe systems. In *2016 IEEE International Solid-State Circuits Conference (ISSCC)*, pages 264–265, Jan 2016.

[76] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014. URL: http://arxiv.org/abs/1409.1556, arXiv:1409.1556.

[77] V. Sze, Y. Chen, J. Emer, A. Suleiman, and Z. Zhang. Hardware for machine learning: Challenges and opportunities. In *2017 IEEE Custom Integrated Circuits Conference (CICC)*, pages 1–8, 2017. doi:10.1109/CICC.2017.7993626.

[78] M. Tan, B. Chen, R. Pang, V. Vasudevan, and Q. V. Le. MnasNet: Platform-aware neural architecture search for mobile. *CoRR*, abs/1807.11626, 2018. URL: http://arxiv.org/abs/1807.11626, arXiv:1807.11626.

[79] K. Ueyoshi, K. Ando, K. Hirose, S. Takamaeda-Yamazaki, J. Kadomoto, T. Miyata, M. Hamada, T. Kuroda, and M. Motomura. QUEST: A 7.49TOPS multi-purpose log-quantized DNN inference engine stacked on 96MB 3D SRAM using inductive-coupling technology in 40nm CMOS. In *2018 IEEE International Solid - State Circuits Conference - (ISSCC)*, pages 216–218, Feb 2018. doi:10.1109/ISSCC.2018.8310261.

[80] S. I. Venieris, A. Kouris, and C.-S. Bouganis. Toolflows for mapping convolutional neural networks on fpgas: A survey and future directions. *ACM Comput. Surv.*, 51(3), June 2018. doi:10.1145/3186332.

[81] S. Vogel, M. Liang, A. Guntoro, W. Stechele, and G. Ascheid. Efficient hardware acceleration of cnns using logarithmic data representation with arbitrary log-base. In *Proceedings of the International Conference on Computer-Aided Design*, ICCAD

'18, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3240765.3240803.

[82] E. Wang, J. J. Davis, P. Y. K. Cheung, and G. A. Constantinides. Lutnet: Learning fpga configurations for highly efficient neural network inference. *IEEE Transactions on Computers*, 69(12):1795–1808, 2020. doi:10.1109/TC.2020.2978817.

[83] E. Wang, J. J. Davis, R. Zhao, H.-C. Ng, X. Niu, W. Luk, P. Y. K. Cheung, and G. A. Constantinides. Deep neural network approximation for custom hardware: Where we've been, where we're going. *ACM Comput. Surv.*, 52(2), May 2019. doi:10.1145/3309551.

[84] P. Wang, P. Chen, Y. Yuan, D. Liu, Z. Huang, X. Hou, and G. Cottrell. Understanding convolution for semantic segmentation. In *2018 IEEE Winter Conference on Applications of Computer Vision (WACV)*, pages 1451–1460, 2018. doi:10.1109/WACV.2018.00163.

[85] S. Wang and P. Kanwar. BFloat16: The secret to high performance on Cloud TPUs. URL: https://cloud.google.com/blog/products/ai-machine-learning/bfloat16-the-secret-to-high-performance-on-cloud-tpus, accessed Jan 19, 2021, published Aug. 24, 2019.

[86] Y. Wang, G. Wei, and D. Brooks. Benchmarking tpu, gpu, and CPU platforms for deep learning. *CoRR*, abs/1907.10701, 2019. URL: http://arxiv.org/abs/1907.10701, arXiv:1907.10701.

[87] Z. Wang, J. Chen, and S. C. H. Hoi. Deep learning for image super-resolution: A survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 1–1, 2020. doi:10.1109/TPAMI.2020.2982166.

[88] Y. Wu and C. T. Huang. Efficient dynamic fixed-point quantization of cnn inference accelerators for edge devices. In *2019 International Symposium on VLSI Design, Automation and Test (VLSI-DAT)*, pages 1–4, 2019. doi:10.1109/VLSI-DAT.2019.8742040.

[89] H. Yonekawa and H. Nakahara. On-chip memory based binarized convolutional deep neural network applying batch normalization free technique on an fpga. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 98–105, May 2017. doi:10.1109/IPDPSW.2017.95.

[90] S. Zeng, G. Dai, H. Sun, K. Zhong, G. Ge, K. Guo, Y. Wang, and H. Yang. Enabling efficient and flexible fpga virtualization for deep learning in the cloud. In *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 102–110, 2020. doi:10.1109/FCCM48280.2020.00023.

[91] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '15, pages 161–170, New York, NY, USA, 2015. ACM. URL: http://doi.acm.org/10.1145/2684746.2689060, doi:10.1145/2684746.2689060.

List of Publications

Journal Papers

1. **K. Ando**, K. Ueyoshi, Y. Oba, K. Hirose, R. Uematsu, T. Kudo, M. Ikebe, T. Asai, S. Takamaeda-Y., and M. Motomura, "Dither NN: Hardware/Algorithm Co-Design for Accurate Quantized Neural Networks," IEICE Transactions on Information and Systems, vol. E102-D, no. 12, pp. 2341-2353, Dec. 2019. doi: 10.1587/transinf.2019PAP0009

2. **K. Ando**, K. Ueyoshi, K. Orimo, H. Yonekawa, S. Sato, H. Nakahara, S. Takamaeda-Y., M. Ikebe, T. Asai, T. Kuroda, and M. Motomura, "BRein Memory: A Single-Chip Binary/Ternary Reconfigurable in-Memory Deep Neural Network Accelerator Achieving 1.4 TOPS at 0.6 W," IEEE Journal of Solid-State Circuits, vol. 53, no. 4, pp. 983-994, Apr. 2018. doi: 10.1109/JSSC.2017.2778702

3. **K. Ando**, S. Takamaeda-Y., M. Ikebe, T. Asai, and M. Motomura, "A Multithreaded CGRA for Convolutional Neural Network Processing," Circuits and Systems, 8, 149-170, 2017. doi: 10.4236/cs.2017.86010

Journal Papers (Coauthored)

1. K. Yamamoto, K. Kawamura, **K. Ando**, N. Metrig, T. Takemoto, M. Yamaoka, H. Teramoto, A. Sakai, S. Takamaeda-Y., and M. Motomura, "STATICA: A 512-Spin 0.25M-Weight Annealing Processor with an All-Spin-Updates-at-Once Architecture for Combinatorial Optimization with Complete Spin-Spin Interactions," IEEE Journal of Solid-State Circuits (JSSC), vol. 56, no. 1, pp. 165-178, Jan. 2021, doi: 10.1109/JSSC.2020.3027702

2. M. Motomura, S. Takamaeda-Y., K. Ueyoshi, **K. Ando**, and K. Hirose, "Examples and Prospect of Deep Neural Network Processor Technology," IEICE-C (Japanese), vol. J103-C, no. 5, pp. 288-297, Mar. 2020. [Translated from Japanese]

3. K. Ueyoshi, **K. Ando**, K. Hirose, S. Takamaeda-Y., M. Hamada, T. Kuroda, and M. Motomura, "QUEST: Multi-Purpose Log-Quantized DNN Inference Engine Stacked on 96-MB 3-D SRAM Using Inductive Coupling Technology in 40-nm CMOS," IEEE Journal of Solid-State Circuits, vol. 54, no. 1, pp. 186-196, Jan. 2019. doi: 10.1109/JSSC.2018.2871623

4. K. Hirose, R. Uematsu, **K. Ando**, K. Ueyoshi, M. Ikebe, T. Asai, M. Motomura, and S. Takamaeda-Y, "Quantization Error-Based Regularization for Hardware-Aware Neural Network Training," IEICE Nonlinear Theory and Its Applications, vol. 9, no. 4, pp. 453-465, Oct. 2018. doi: 10.1587/nolta.9.453

International Conference Proceedings

1. **K. Ando**, K. Ueyoshi, Y. Oba, K. Hirose, R. Uematsu, T. Kudo, M. Ikebe, T. Asai, S. Takamaeda-Y., and M. Motomura, "Dither NN: An Accurate Neural Network with Dithering for Low Bit-Precision Hardware," The 2018 International Conference on Field-Programmable Technology (FPT'18), pp. 6-13, Dec. 2018.

2. **K. Ando**, K. Ueyoshi, K. Orimo, H. Yonekawa, S. Sato, H. Nakahara, M. Ikebe, S. Takamaeda-Y., T. Asai, T. Kuroda, and M. Motomura, "In-Memory Area-Efficient Signal Streaming Processor Design for Binary Neural Networks," 2017 IEEE 60th International Midwest Symposium on Circuits and Systems (MWSCAS), Aug. 2017.

3. **K. Ando**, K. Ueyoshi, K. Orimo, H. Yonekawa, S. Sato, H. Nakahara, M. Ikebe, T. Asai, S. Takamaeda-Y., T. Kuroda, and M. Motomura, "BRein Memory: A 13-Layer 4.2k Neuron/0.8M Synapse Binary/Ternary Reconfigurable In-Memory Deep Neural Network Accelerator in 65nm CMOS," 2017 Symposium on VLSI Circuits (VLSI), pp. C24-C25, Jun. 2017.

4. **K. Ando**, K. Ueyoshi, K. Orimo, M. Ikebe, S. Takamaeda-Y., T. Asai, and M. Motomura, "Throughput Analysis of a Data-Flow Reconfigurable Array Architecture for Convolutional Neural Networks," RIEC International Symposium on Brain Functions and Brain Computer, Feb. 2017.

5. **K. Ando**, K. Orimo, K. Ueyoshi, M. Ikebe, T. Asai, and M. Motomura, "Reconfigurable Processor Array Architecture for Deep Convolutional Neural Networks,"

Workshop on Synthesis And System Integration of Mixed Information Technologies (SASIMI), Oct. 2016.

International Conference Proceedings (Coauthored)

1. J. Suzuki, **K. Ando**, K. Hirose, K. Kawamura, T. V. Chu, M. Motomura, and J. Yu, "ProgressiveNN: Achieving Computational Scalability without Network Alteration by MSB-first Accumulative Computation," International Symposium on Computing and Networking (CANDAR), Nov. 2020.

2. K. Shiba, T. Omori, K. Ueyoshi, **K. Ando**, K. Hirose, S. Takamaeda-Y., M. Motomura, M. Hamada, and T. Kuroda, "A 3D-Stacked SRAM Using Inductive Coupling with Low-Voltage Transmitter and 12:1 SerDes," IEEE 2020 International Symposium on Circuits and Systems (ISCAS), pp. 1-5, Oct. 2020.

3. K. Yamamoto, **K. Ando**, N. Mertig, T. Takemoto, M. Yamaoka, H. Teramoto, A. Sakai, S. Takamaeda-Y., and M Motomura, "STATICA: A 512-Spin 0.25M-Weight Full-Digital Annealing Processor with a Near-Memory All-Spin-Updates-at-Once Architecture for Combinatorial Optimization with Complete Spin-Spin Interactions," 2020 IEEE International Solid-State Circuits Conference (ISSCC), pp. 138-139, Feb. 2020.

4. Y. Oba, **K. Ando**, T. Asai, M. Motomura, S. Takamaeda-Y., "DeltaNet: Differential Binary Neural Network," IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP), p. 39, July 2019.

5. T. Kudo, K. Ueyoshi, **K. Ando**, K. Hirose, R. Uematsu, Y. Oba, M. Ikebe, T. Asai, M. Motomura, S. Takamaeda-Y, "Area and Energy Optimization for Bit-Serial Log-Quantized DNN Accelerator with Shared Accumulators," IEEE 12th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC), pp. 237-243, Sep. 2018.

6. R. Uematsu, **K. Ando**, K. Ueyoshi, K. Hirose, M. Ikebe, T. Asai, S. Takamaeda-Y., and M. Motomura, "Exploring CNN Accelerator Design Space on a Dynamically Reconfigurable Hardware Platform," Workshop on Synthesis And System Integration of Mixed Information Technologies (SASIMI), Mar. 2018.

7. K. Ueyoshi, **K. Ando**, K. Hirose, S. Takamaeda-Y., J. Kadomoto, T. Miyata, M. Hamada, T. Kuroda, and M. Motomura, "QUEST: A 7.49-TOPS Multi-Purpose Log-Quantized DNN Inference Engine Stacked on 96MB 3D SRAM using Inductive-Coupling Technology in 40nm CMOS," 2018 IEEE International Solid-State Circuits Conference (ISSCC), pp. 216-218, Feb. 2018.

8. S. Takamaeda-Y., K. Ueyoshi, **K. Ando**, R. Uematsu, K. Hirose, M. Ikebe, T. Asai, and M. Motomura, "Accelerating Deep Learning by Binarized Hardware," 2017 Asia-Pacific Signal and Information Processing Association Annual Summit and Conference (APSIPA ASC), pp. 1045-1051, Dec. 2017.

9. K. Hirose, **K. Ando**, K. Ueyoshi, M. Ikebe, T. Asai, M. Motomura, and S. Takamaeda-Y., "Quantization Error-based Regularization in Neural Networks," SGAI International Conference on Artificial Intelligence (SGAI), Dec. 2017.

10. K. Hirose, R. Uematsu, **K. Ando**, K. Orimo, K. Ueyoshi, M. Ikebe, T. Asai, S. Takamaeda-Y., and M. Motomura, "Logarithmic Compression for Memory Footprint Reduction in Neural Network Training," 2017 Fifth International Symposium on Computing and Networking (CANDAR), pp. 291-297, Nov. 2017.

11. K. Hirose, R. Uematsu, **K. Ando**, K. Ueyoshi, M. Ikebe, T. Asai, M. Motomura, and S. Takamaeda-Y., "A Regularization Approach for Quantized Neural Networks," International Workshop on Highly Efficient Neural Networks Design (HENND), Oct. 2017.

12. K. Ueyoshi, **K. Ando**, K. Orimo, M. Ikebe, T. Asai, and M. Motomura, "Exploring Optimized Accelerator Design for Binarized Convolutional Neural Networks," 2017 International Joint Conference on Neural Networks (IJCNN), pp. 2510-2516, May 2017.

13. K. Orimo, **K. Ando**, K. Ueyoshi, M. Ikebe, T. Asai, and M. Motomura, "FPGA Architecture for Feed-Forward Sequential Memory Network Targeting Long-Term Time-Series Forecasting," 2016 International Conference on ReConFigurable Computing and FPGAs (ReConFig), pp. 1-6, Nov. 2016.