Received: 28 March 2020 | Revised: 13 June 2020 | Accepted: 2 July 2020

DOI: 10.4218/etrij.2020-0125

SPECIAL ISSUE

ETRI Journal WILEY

Automated optimization for memory-efficient high-performance deep neural network accelerators

HyunMi Kim | **Chun-Gi Lyuh** | **Youngsu Kwon**

AI SoC Research Division, Electronics and Telecommunications Research Institute, Daejeon, Rep. of Korea

Correspondence
HyunMi Kim, AI SoC Research Division, Electronics and Telecommunications Research Institute, Daejeon, Rep. of Korea.
Email: chaos0218@etri.re.kr

Funding information
This research was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government(MSIT) (No. 2018-0-00195, Artificial Intelligence Processor Research Laboratory).

The increasing size and complexity of deep neural networks (DNNs) necessitate the development of efficient high-performance accelerators. An efficient memory structure and operating scheme provide an intuitive solution for high-performance accelerators along with dataflow control. Furthermore, the processing of various neural networks (NNs) requires a flexible memory architecture, programmable control scheme, and automated optimizations. We first propose an efficient architecture with flexibility while operating at a high frequency despite the large memory and PE-array sizes. We then improve the efficiency and usability of our architecture by automating the optimization algorithm. The experimental results show that the architecture increases the data reuse; a diagonal write path improves the performance by 1.44× on average across a wide range of NNs. The automated optimizations significantly enhance the performance from 3.8× to 14.79× and further provide usability. Therefore, automating the optimization as well as designing an efficient architecture is critical to realizing high-performance DNN accelerators.

KEYWORDS
accelerators, architecture, automation, deep neural network (DNN), optimization

1 | INTRODUCTION

Deep neural networks (DNNs) have recently become the most influential technology for diverse artificial-intelligence (AI) applications [1]. The scope of AI applications that use DNNs has been extended to speech recognition [2], complex computer games [3], disease diagnosis [4], and autonomous vehicles [5], with considerable success in image classification, object recognition, and computer vision [6]. In addition, DNNs continue to evolve in complexity beyond human accuracy. The remarkable abilities of DNNs include high computational capability and high energy (power) consumption along with the availability of massive data.

Research on DNNs advanced in the early phase by using general-purpose computing processors, such as central processing units (CPUs) and graphics processing units (GPUs). However, domain-specific accelerators for DNNs are increasingly required, as recent research show that DNN-specific accelerators surpass CPUs and GPUs in terms of performance and energy efficiency [7–13].

Applications such as datacenters leverage more general high-performance DNN accelerators, which can efficiently process various NNs, while mobile and deeply embedded applications such as intelligent Internet-of-Things systems focus on highly specialized accelerators to achieve high energy efficiency [14]. The high-performance DNN accelerator, which we target in this paper, involves programmability to process various NNs, enables the efficient implementation of large memory and many processing elements (PEs), and provides a software stack such as a complier with highly optimized algorithms to effectively operate the accelerator.

Accordingly, we take the first step toward presenting a memory-efficient architecture and automated optimization algorithm for high-performance general DNN accelerators. Our accelerator implementation operates at 1 GHz using a simple data connection despite its large buffer size and operation unit. Furthermore, we increase the data reuse by leveraging flexible internal memory, programmable data control, and a memory-allocation algorithm, thereby maximizing the performance of our accelerator along with automating hardware scheduling.

The rest of this paper is organized as follows. In Section 2, we delineate neural networks (NNs) and the structure of DNN accelerators. We propose a flexible and programmable architecture for efficient DNN accelerators in Section 3, and automating optimization algorithms are introduced in Section 4. Next, in Section 5, we present the evaluation results, and we conclude this study in Section 6.

2 | BACKGROUND

2.1 | Neural networks

Various types of NNs have been developed depending on the applications. Among them, the most widely used NNs can be categorized as multi-layer perceptrons (MLPs), convolutional neural networks (CNNs), and recurrent neural networks (RNNs). Notably, CNNs are utilized to extract a spatial feature in image-domain applications, such as object detection, while RNNs are well-suited for time-series prediction applications such as speech recognition and natural language processing. MLPs are useful to solve stochastic problems that involve approximation. However, for more complex problems, hybrid NNs such as convolutional RNNs [15] and a combination of multiple NNs [16] are also applied to improve the accuracy.

The NN architecture is composed of a directed acyclic graph (DAG) that comprises a stack of multiple types of layers, such as a convolutional layer (CONV), fully connected layer (FC), non-linear activation layer (ACTV), pooling layers (POOL), among others. FC and CONV layers generate the output vector ($\mathbf{H}(x)$) by performing a multiply-accumulate (MAC) between the input vector ($\mathbf{x}$) and weight matrix ($\mathbf{W}$), followed by an element-wise addition (EW-ADD) of the bias vector ($\mathbf{b}$) as follows:

$$\mathbf{H}(x) = \mathbf{W}x + \mathbf{b}. \tag{1}$$

where CONV treats two-dimensional (2D) feature maps (*fmaps*) with width (ie, W_i) and height (ie, H_i), as depicted in Figure 1, and FC handles scalars, for an element of weight matrix and

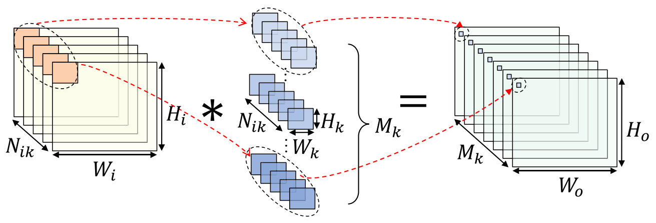


FIGURE 1 CONV Computation. N_{ik} input *fmaps* of size $W_i \times H_i$ are convolved with N_{ik} *weights* of size $W_k \times H_k$ using a sliding window. In addition, CONV generates an output *fmap* of size $W_o \times H_o$. This process is repeated M_k times with different weights. Consequently, M_k output *fmaps* of size $W_o \times H_o$ are produced. The W_o or H_o is calculated using parameters such as padding or stride size

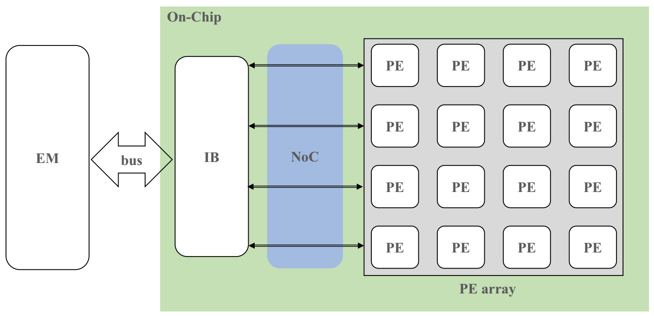


FIGURE 2 Architecture of the DNN accelerator

input vector. ACTV applies non-linear operations, such as rectified linear unit, sigmoid, and hyperbolic tangent, to the output of the CONV or FC layer. POOL, which is optionally applied after CONV, processes by selecting a maximum value or an average value of the values within the predetermined 2D area and reduces the sample number of *fmaps*. In addition to the widely used layers, the training performance can be enhanced using techniques such as normalization and regularization, which are processed via element-wise operations.

2.2 | Structure of DNN accelerators

Domain-specific accelerators for DNN have been proposed to achieve higher processing performance and better energy efficiency than those achieved using conventional processors such as CPU and GPU. As depicted in Figure 2, the high-level block diagram of a DNN accelerator comprises a large external memory (EM) such as DRAM, high-bandwidth internal buffer (IB), and computation engine that consists of an array of PEs. An EM is essential for large-scale applications such as DNNs, as the amount of activation data and weights of layers is large and deeper NNs increase both the amount of data and weight of the layers. However, DNN accelerators incorporate high-bandwidth IBs because of high-performance requirements. In addition, data-reuse techniques that utilize IBs improve both energy efficiency and performance, as EM access requires energy consumption of a higher order than that required by on-chip buffers [9]. A PE array with high parallelism executes operations such as MAC to process the NNs mentioned in Section 2.1. Another important microarchitecture for DNN accelerators is the network-on-chip (NoC) design between IB/PEs and PEs. The NoC design with a dataflow style considerably affects the performance in terms of both throughput and latency.

2.3 | Efficient DNN accelerators

The efficiency of DNN accelerators is evaluated on the basis of both cost (ie, area and power consumption) and performance (ie, throughput and latency). IB structures that support flexible data layout improve the efficiency by maximizing the data reuse and minimizing the power consumption. Low-complexity NoC structures reduce cost and support high bandwidths by implementing high frequencies. In addition, the design of a small PE decreases the total area of a PE array, as PE is replicated as required by the target operation performance. Additionally, the programmable data access eliminates a redundant data copy such as im2col-based CONV, thereby reducing memory consumption and the data bandwidth required between EM and IB. Another crucial factor for performance is dataflow, which describes the manner of dispatching and fetching operands to PEs.

In addition, hardware-dependent optimization algorithms are indispensable along with an efficient hardware architecture. The data-layout decision enhances the efficiency by maximizing the data reuse in IB or PE array, as IB and PE arrays can accommodate the total size of the activation data and weights for DNNs. Maximizing the data reuse significantly affects both performance and cost, as data movement between IB and EM consumes considerable processing time and high power. Although the data reuse is maximized in IB by the efficient data-layout decision, data are still moved between IB and EM because IB is insufficient to store all the data for DNNs. Therefore, efficient scheduling increases the throughput by hiding the data-movement time. Finally, the DNN accelerator is easily leveraged in various applications by automating these algorithms.

3 | ARCHITECTURE

We propose an architecture that includes the IB structure, dataflow path between IB and PEs, and dataflow controller for memory-efficient high-performance DNN accelerators. The architecture aims to operate at a high frequency with a large size of the PE array and IB for implementing

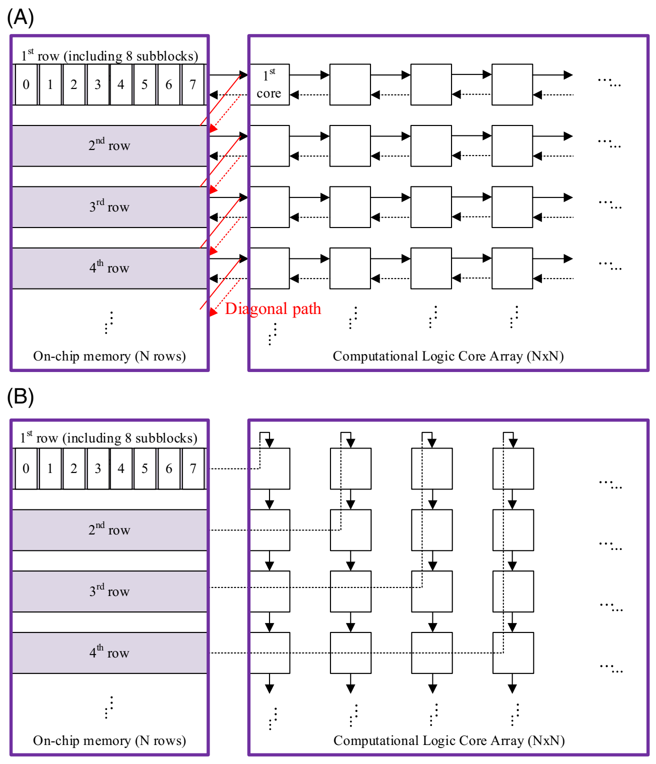


FIGURE 3 Proposed IB structure for DNN accelerators: (A) *fmap* dataflow including the diagonal path and (B) weight dataflow. The solid and dashed lines represent the input and output *fmap*, respectively, and the red line represents the diagonal path in (A). The dashed line in (B) represents the weight dataflow and is implemented via a feed-through way in our design

high-performance processors. We design an architecture based on a short datapath to operate at a high frequency for high-performance accelerators. Additionally, the architecture based on the output-stationary dataflow, which reuses the output within a PE during a computation, is adopted to enable various high-performance operations via a technique such as fusing operations as well as an efficient MAC operation.

3.1 | Buffer structure and dataflow

In Figure 3, we depict the efficient large-scale IB structure and the connection paths with a PE array which is a set of systolic-arrays [17] based computational cores. IB comprises a set of memory rows that are aligned with and arranged in each row of the PE array. IB rows are implemented with single-port SRAMs, which is called the sub-block to reduce the chip area as shown in Figure 3. Only the datapath between the corresponding buffer and PE array rows are connected for input and output activations. Exceptionally, weights are fed from a buffer row to the first PE core of the corresponding PE array column as shown in Figure 3B. Each IB row stores the loaded input activations and weights from EM prior to performing a computation. When starting a computation, input activations in an IB row are fed to the first PE core of the corresponding PE array row, as depicted by the solid arrows in Figure 3A. Similarly, the weights in an IB row enter the first core of the corresponding PE array column, as depicted in Figure 3B. The input activations in the first core of the PE array row are horizontally transferred, and the weights in the first core of the PE array column flow vertically until the data and weights arrive at the last core of the PE array row or column, respectively. The PE cores process operations for DNNs, by using systolic input and weight supplies. Because this architecture supports the output-stationary dataflow, the necessary data are continuously supplied until the final output is calculated. For example, for convolution operations, whole 3D input tensor data and weights fed to PEs with the proposed short datapath and the output register within the PE core store the temporary and result data by an output-stationary architecture. After finishing a computation of a PE core, as depicted by the dashed lines of Figure 3A, the output is delivered from right to left in a row of the PE array in a systolic manner and eventually stored to the corresponding IB row. The PE core includes an adder and a multiplier for computation such as MAC, element-wise operation, rectified linear unit (ReLu), and even pooling, a few data registers for temporarily storing data or transferring data between PEs, and several multiplexers and various datapath between registers and arithmetic elements for the configuration of the target operation.

In addition to the basic dataflow using the row and column connection structure each for activation data and weights, the diagonal path for the input and output activation dataflow, which is denoted by the red lines, is introduced between an adjacent buffer and a PE row (notably, the diagonal path connects a PE-array row, and the buffer row is located diagonally below the PE-array row) for efficient operations, especially for operations with high data-reuse rates such as CONV. The diagonal datapath prevents the input activation duplication in multiple buffer rows and rearranges the output activation between buffer rows, allowing for a padding sand kernel shape of the operation, which will be processed in the next layer. For example, as depicted in Figure 4, a $W_i \times 2$ input fmap (ie, *ifmap*) can be assigned to a buffer row when processing the 2D CONV of $W_i \times H_i$ ifmap by using three 3×3 kernels with a zero-padding size of 1. Notably, the 3×3 kernels are stored in separate sub-blocks to that for ifmap. To calculate the first output row, the I1–I6 input data are read from buffer row 0 and fed to the first PE row by using the corresponding row path (ie, the black bold line in Figure 3A), while the diagonal path is used to feed the I7–I9 ifmap to the first PE array row. To access data, the addresses for the target input and weight data are generated cycle by cycle in synchronization. The padding and diagonal path area are also identified, and their special signals are made in the data-access controller (DAC),

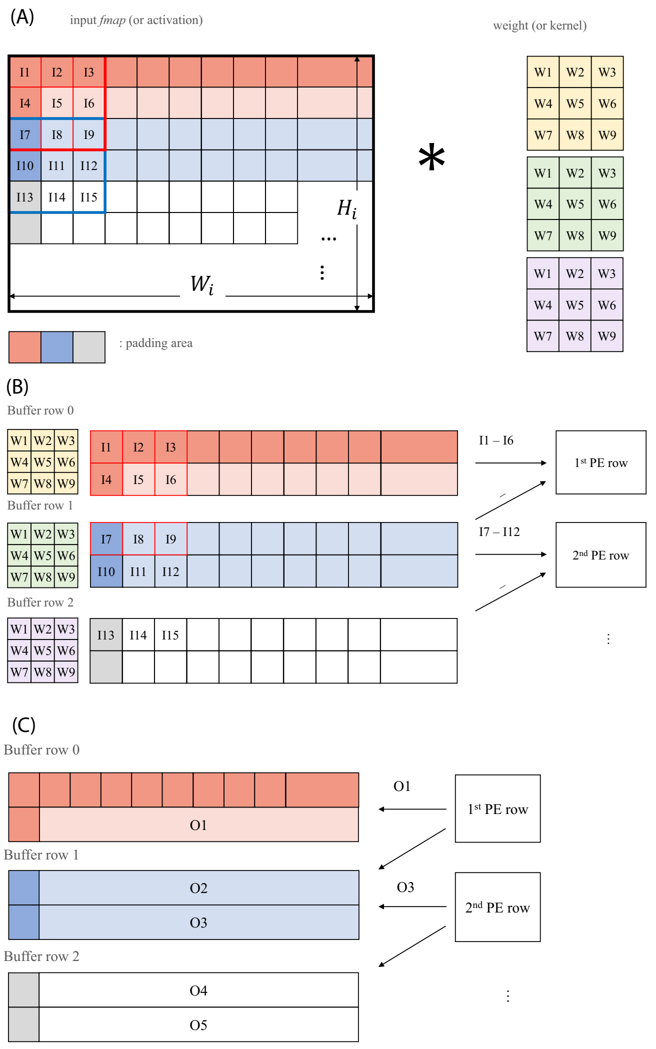


FIGURE 4 Example of buffer allocation and dataflow: (A) example convolution parameters ($W_i \times H_i$ *ifmap* and three 3×3 kernels), (B) buffer allocation of *ifmap* and weights, and *ifmap's* data flow, and (C) buffer allocation and dataflow of the output *fmap*

as will be described in Section 3.2. If the padding size of the next processed CONV is 1, the output addresses are calculated, allowing for the padding area. In addition, the diagonal write path is active when the second row of output fmap (ie, O2) is stored in the second buffer row from the first PE row. This proposed scheme eliminates the redundant data stored in the buffer and enables data reordering during the computation stage, without an additional data-reordering stage or data movement so that it can improve memory efficiency and the data-reuse rate. Each amount of three data components, that is, input, output activations, and weights, significantly varies depending on the NN structure and an IB row stores all the components that are simultaneously accessed during a computation stage despite implementing with single-port SRAM. Therefore, determining the number of sub-blocks for a buffer row is important for flexibility, concurrent accessibility, and the chip area. We note that the use of smaller buffers increases the total chip area. We will evaluate the impact of the number of sub-blocks in Section. 5.

3.2 | Data-access controller

To access (ie, read and write) a buffer row for the abovementioned three data components during a computation stage, the following three DACs are proposed for each of the three components: DAC-I for input activations, DAC-W for weights, and DAC-O for output activations. DACs control the dataflow by generating a sequence of read/write addresses every cycle. A DAC comprises a programmable N-dimensional nested-loop based address generator. The programmable parameters include a repeat count and an incremental offset for each loop dimension, as well as an initial address. The number of dimensions, which corresponds to the number of changed directions of the accessed data within a tensor, is decided by analyzing the accessed data order of various NNs. For example, for the operation that fuses a convolution layer and pooling layer (CONVPOOL), DAC-I adopts the seven-level nested-loop structure to support seven dataflow directions: width, height and channel directions of the kernel data, horizontal and vertical directions for pooling, and two sliding-window directions. The address of the accessed data at a single cycle is generated using Algorithm 1.

Algorithm 1: *n*-level nested Do-loop based address-generator algorithm

```
Do $i_1 =$ 0 to $u_1$
    ...
    Do $i_n =$ 0 to $u_n$

        address $= \text{addr}_0 + \sum_{k=1}^{n} i_k \times \text{inc}_k$

    EndDo $i_n$
    ...
EndDo $i_1$
```

where addr_0 denotes the initial address and u_x and inc_x the total repeated count and index increments of each loop, respectively. The programmable DACs are designed only using adders and multiplexers instead of expensive multipliers, and the generated addresses are transferred in a systolic manner from the first to last IB row. The accessed data by the generated addresses are synchronized with the PE configuration signal for the target computation and fed to the PE array. Additionally, DAC-I and DAC-O contain the diagonal path control to access the IB row below and the padding control to process the padding area using the generated address and supplementary information, such as padding/stride sizes and indicators of which loop level represents which direction component.

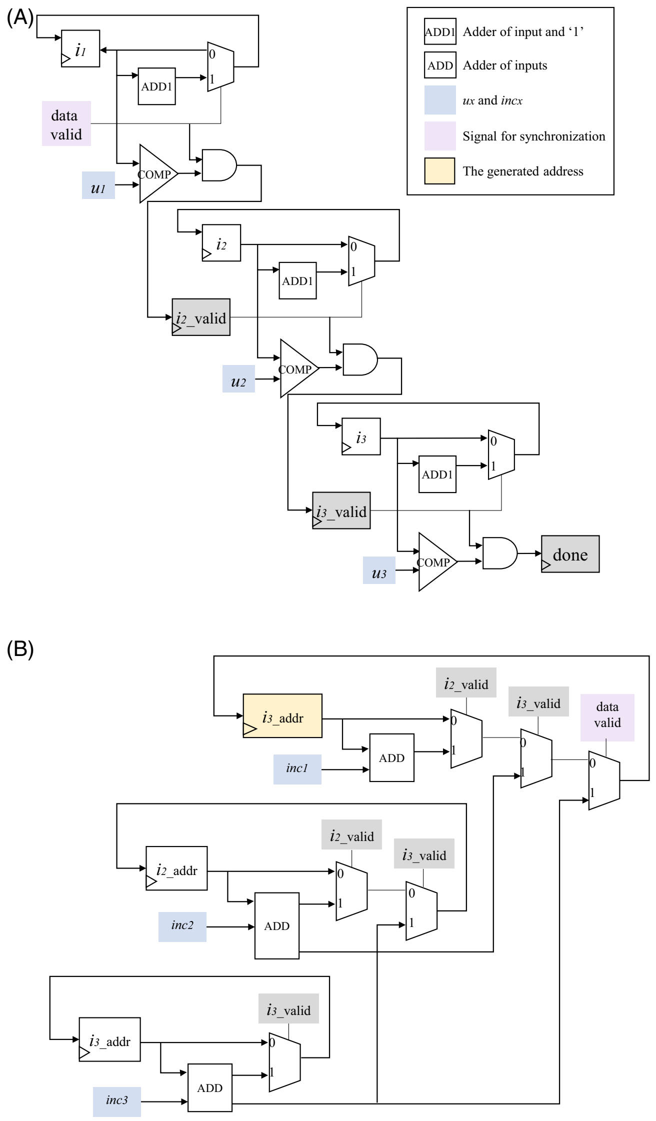


FIGURE 5 Architecture of 3-level nested Do-loop based address-generator: (A) loop control block and (B) address generator block

Figure 5 depicts the architecture for a three-level address generator. i_x and i_x_addr are set by zero and the initial address addr_0, respectively, before operating DAC. The address of the accessed data in next time is generated in synchronization with the data valid signal (ie, light purple box). Since PEs perform the various operations that require different data components, the data valid signal supports the condition that the required and accessed data is synchronized with the PE configuration signals for a target operation. The blue boxes are the registers for programmability and the generated address is stored in the yellow box. The loop control block of Figure 5A controls the incremental state of the Do-loop in Algorithm 1 and generates the valid signals in the loop level that are colored by the grey. The loop valid signals are transferred to the address generator block of Figure 5B and select the valid address.

4 | AUTOMATED OPTIMIZATION

The optimizations are described as the key technology that maximizes the efficiency of the DNN accelerator by using the proposed memory-efficient structure and dataflow-control scheme, in this section. First, the abovementioned three data components are allocated in a buffer row to efficiently exploit the flexible buffer. Second, the parameters are defined for optimal scheduling, which maximizes a throughput. Third, the automation technique for deciding the optimal data layout and scheduling is proposed to increase the performance of the proposed accelerator for various NNs. Finally, fusing operations are introduced to improve the data-reuse rate by removing the processing time required for the data movement.

4.1 | Optimization of buffer allocation

For the efficiency of the proposed IB and PE array architectures and data-access scheme, the optimization algorithm adopts three techniques including slicing and tiling for activation, and flexibly allocating three data components in multiple buffer sub-blocks. In this section, we describe the buffer-allocation optimization for the CONVPOOL operation that deals with 3D tensors, as an example.

First, an input 3D tensor that contains a padding area is horizontally sliced at the same height (ie, slice height), and a slice is allocated in a buffer row, as depicted in Figure 6 for the row-wise IB structure. The sliced and allocated input tensors are fed and exploited as an operand for parallel computing. Because of the row-wise datapath including the diagonal path, the slice height is decided by the kernel height, vertical kernel stride size, vertical pooling size, and vertical pooling stride. Specifically, a slice height is decided to satisfy the following two conditions.

- The slice height is set as a multiple of (the vertical stride size of pooling × the stride vertical size of the convolution kernel).
- (slice height × 2) is greater than/equal to (kernel height–1).

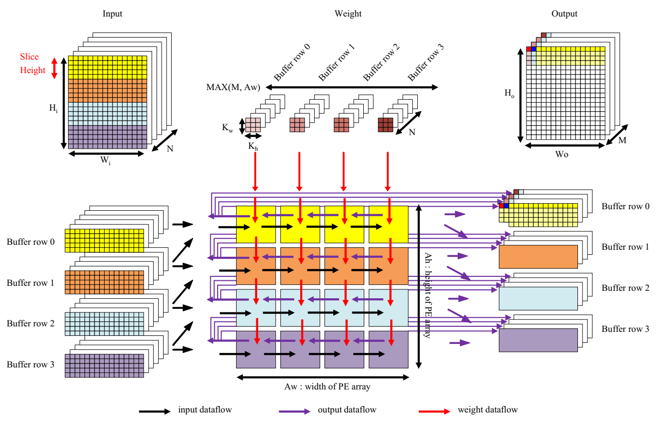


FIGURE 6 Example of *fmap* sliced by the slice height, buffer allocation, and dataflow in the architecture that comprises a 4 × 4 PE array and 4 buffer rows

The batch size can be used as an additional factor to decide the slice height because one batch can be allocated to multiple buffers.

When the capacity of IB is insufficient to store the total data including input, output, and weight of one convolution layer, a tensor is vertically tiled. The tiled input tensor is loaded from EM to IB after slicing so that the buffer capacity is validated with the tensor piece size, which is determined by the combination of tiling and slicing. This procedure is repeated until the buffer row can accommodate a tensor piece. The output tensor is also stored in the buffer row after finishing a computation in the PE array. If IB is sufficient to store input, output, and weights without tiling, the output is reused in IB for the next operation. Otherwise, the output tile is temporarily stored in IB rows and loaded out to EM to secure the memory space for processing another tile. Unlike tensor data, the weights of a layer are tiled on the basis of the number of columns in the PE array (ie, A_w in Figure 6), as the weights are passed in column-wise. Assuming that the size of the PE array is $A_w \times A_h$ (ie, 4 × 4 in Figure 6) and that the number of weights for a layer is M, one weight tile includes the number of max (M, A_w) and the number of weight tiles is M/A_w.

Finally, an input slice, an output slice, and a weight tile are flexibly allocated in the sub-blocks of a buffer row, allowing for simultaneous buffer access. During the computation phase, an input slice and a weight tile are concurrently read from a buffer row and transferred to the first core of the corresponding row and column, respectively, as two operands. Therefore, the buffer sub-blocks for an input slice and a weight tile are separated to operate without latency for synchronization. This is because if sub-blocks for input and weights are integrated, the accelerator requires an additional processing time (latency) to synchronize between input and weight feedings. However, the sub-block separation for the input and output slices is optionally determined by comparing the performance during the automated optimization. The separated sub-block allocation improves throughput while the integrated sub-block allocation increases the buffer utilization by removing buffer fragmentation.

4.2 | Scheduling of hardware operations

The scheduling of hardware operations is one of the most significant optimization steps for performance maximization. The hardware operations are defined as input-loading (IL), weight-loading (WL), computation (CT), and output-storing (OS) for scheduling. The performance of the four operations is first calculated by exploiting the results of the buffer allocation, as each hardware operation time relies on the processed data size. With the calculated performance of each operation, search spaces are explored. The search spaces are determined using a combination of the following condition parameters.

Cond. 1. The capacity of weights for the overall NN.

Cond. 2. The availability of double buffering for each component.

Cond. 3. The separation/integration of input and output buffers.

Cond. 4. Input and output data reuse.

For example, the initial search space for each layer is set to 24 if only one data for every three data components are present. When the weights for the overall NN are acceptable in an IB with a spare buffer space for input and output activations, the number of search spaces is 2^3 (ie, 8), where input double buffering or not = 2 options, output double buffering or not = 2 options, and input and output buffers separation or integration = 2 options, thereby totaling to $2 \times 2 \times 2 = 8$ options. However, when IB is insufficient to store the overall weights, the number of search spaces is 2^4 (ie, 16) by adding a weight double-buffering parameter.

After setting the number of the initial search spaces to 24, they are reduced by checking the input and output data reuse in IB between the processed layers, as the double buffering parameter is removed if the data are reused in IB. The final search spaces are explored on the basis of a pre-defined scheduling model by using the dependency between hardware operations. In Figure 7, we depict an example of scheduling models, where the output and weight exploit double buffers. The separation and integration options of the input and output buffers indicate the different CT performance since simultaneous access is impossible. Therefore, the lighter purple boxes indicate a lower performance than those of the darker purple boxes due to the use of the integrated input and output buffer 0. Although the separated buffer option consumes the shorter operation time when processing the same amount of data, the integrated buffer option can lead to whole performance improvement, especially for the target NN which processes large data, by increasing the buffer utilization.

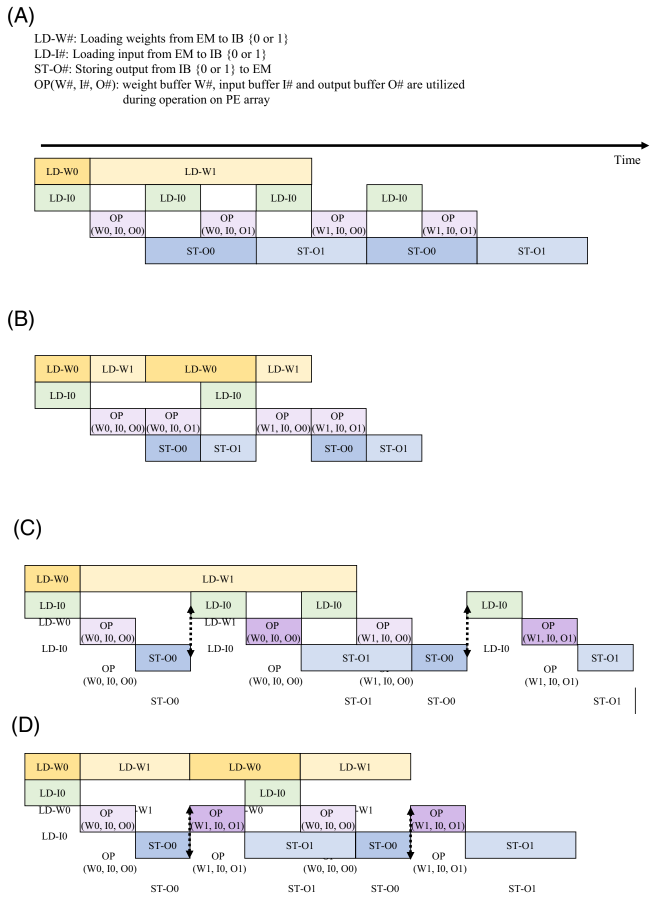

(A)
LD-W#: Loading weights from EM to IB {0 or 1}
LD-I#: Loading input from EM to IB {0 or 1}
ST-O#: Storing output from IB {0 or 1} to EM
OP(W#, I#, O#): weight buffer W#, input buffer I# and output buffer O# are utilized during operation on PE array
Time
LD-W0
LD-W1
LD-I0
OP (W0, I0, O0)
OP (W0, I0, O1)
OP (W1, I0, O0)
OP (W1, I0, O1)
ST-O0
ST-O1
(B)
(C)
(D)

FIGURE 7 Scheduling example of a search space with double buffers for output and weight when two input tiles and two weight tiles are processed: (A) weight-loading priority and separated in/out buffers, (B) input-loading priority and separated in/out buffers, (C) weight-loading priority and integrated in/out buffer, and (D) input-loading priority in/out buffer

4.3 | Automating optimizations

Automating the optimization algorithms is necessary to apply various NNs to the designed DNN accelerator. The automation algorithm includes the parameterization of search spaces, the cost model, and a searching method. Specifically, a searching method allows for the number of the handled data for a target operation, the latency hiding technique, and the data-reuse methods. The automated optimization aims to maximize the data reuse in IB-level after optimizing the PE-level data reuse in a prior stage such as fusing operations in Section 4.5. The number of the handled data is also decided during the PE-level data-reuse stage. The weight data-reuse is searched by verifying the IB capacity for the whole weights for a target NN because weights are used regardless to test inputs. Since the latency hiding is realized by using double buffering, the search spaces include the availability of double buffering in each data component. The additional exploration space for our architecture is the separation possibility between input and output buffers for an efficient sub-block utilization. The buffer allocation search spaces including tiling, slicing, IB-level data reuse between operations are combined with the pre-determined search spaces. During optimization, the automation algorithm validates the search spaces that are the combination of the optimization parameters for buffer allocation and the scheduling search spaces, following which it calculates the cost. The space with the best cost is decided as the final space by comparing the costs of all the available spaces with one another. We exploit the hardware behavior modeling to compute the cost. The scheduling model is parameterized using the cost of four operations, that is, IL, WL, CT, and OS, and the cost is calculated based on the hardware behavior modeling and the data size decided via buffer allocation. Therefore, the final cost for a condition is yielded with the hardware model and the scheduling model. A summary of the automated optimization process is presented in Algorithm 2. Additionally, the cost of each hardware operation is weighted to allow for variations in the hardware implementations. The weights can be updated to apply the practical hardware operation model via an experiment in the real environment.

Algorithm 2: Automated optimization process

```
 Step 1: Decide Search Spaces
 Define search elements
    $X_1 = SP_{io}$
    $X_2 = \prod_{i=1}^{N_i} DB_i$
    $X_3 = \prod_{o=1}^{N_o} DB_o$
    $X_4 = \prod_{w=1}^{N_w} DB_w$
 Set search spaces to load all the weights of the target NN
    $S_{\text{allwgt}} = \prod_{x=1}^{3} X_x$
 Set search spaces to partially load necessary weights
    $S_{\text{partwgt}} = \prod_{x=1}^{4} X_x$
 Set total search spaces
    $S = S_{\text{allwgt}} \cup S_{\text{partwgt}}$
 Step 2: Explore Buffer Allocation
 Initial best cost = maximum, $i = 0$
 Repeat $s_i \in S$
    Initial number of tile = 1
    Repeat vertically tiling (Tiling)
        Calculate the slice height (Slicing)
        Validate the sliced tensor fits to buffer (Valid)
        If Valid equals to 1
            Calculate the cost based on hardware modeling
            If the cost < the best
                Update the best cost
                Store the current search space & buffer allocation
            If n(tiles) equals to 1, break further tiling
        Increase the number of tiles
    until minimum tile width
    $i = i + 1$
 until $i$ equals n($S$)
```

where N_i, N_o, and N_w denote the number of the processed data for input, output, and weight components, respectively, and *SP* represents whether the buffer structure for input and output is unified or separated. The term *DB* indicates whether double buffering is used, and its value is set to 1 if the data are reused between the processed operations.

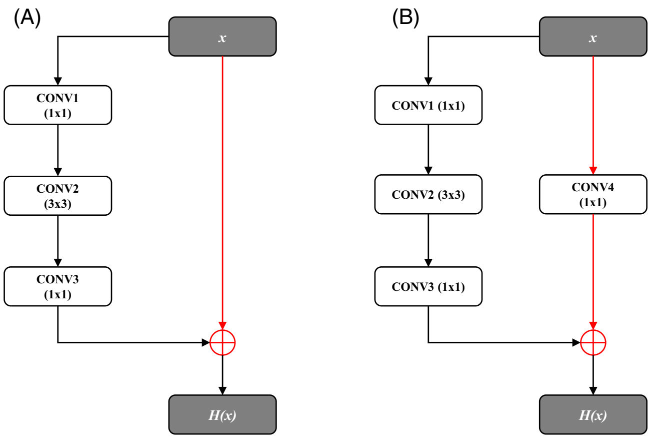


FIGURE 8 Structure of the basic blocks for *ResNet*: (A) residual block and (B) identity block. The red line denotes the residual (skip or shortcut) connection

In addition to Algorithm 2, the batch size can be combined with search spaces during exploratory buffer allocation as a condition since the amount of handled data is different by the batch size, and impacts on the PE array utilization and IB-level data reuse. Therefore, batch size searching can be also added to the automated optimization. We have elucidated the optimization results in Section 5.3, including the batch size searching algorithm.

4.4 | Fusing operations

The fusing operations of DAGs are one of the essential factors to improve the efficiency of DNN accelerators, as fused operations are processed without an additional data movement between the PE array and IB or between IB and EM. In this section, we describe the processing of the fusing operation based on the proposed architecture. We previously discussed the data control for CONVPOOL, which fuses CONV and POOL, as an example in Section 3.2. Therefore, we will discuss the fusing of CONV and the residual connection (RES), also called shortcut or skip connection, which is used in ResNet [18] as another example.

ResNet, which is one of the widely used NNs, introduces a RES to build the residual blocks and identity blocks to increase the image-recognition accuracy, as depicted in Figure 8. RES operates by adding output activations from two layers in an element-wise manner. Because our architecture uses output-stationary dataflow, the output of CONV3 is stored in a register file of PEs after the convolution operation, and then *x* in IB is fed to PEs for EW-ADD. For the processing of the fused CONV and RES (CONVRES), *x* can be loaded out to EM or stored in IB after processing CONV1 according to the IB state. Therefore, the parameters for input *x* of CONVRES are

TABLE 1 Layer structure and parameters of the evaluated NNs

NNs	CONVs	FCs	POOLs	RESs	Total weights	Max. weights[a]	Max. fmaps[a,b]	# of MACs
Alexnet [19]	5	3	6	N/A	62.4 M	37.8 M	0.2 M	724 M
VGG-16 [20]	13	3	5	N/A	138.4 M	102 M	4 M	15.3 B
ResNet-50 [18]	50	N/A	2	16	23.8 M	2.4 M	2.1 M	2.5 B
Yolov2-416 [21]	23	N/A	5	N/A	50.9 M	11.8 M	6 M	14.7 B
Yolov3-416 [22]	75	N/A	N/A	23	61.9 M	4.7 M	8.3 M	5.1 B
SqueezeNet [23]	26	N/A	3	N/A	1.2 M	0.5 M	1.3 M	1.7 B

[a]Max. Weights and fmaps are the maximum required memory footprint for weights and fmaps data, respectively, when processing layer by layer.

[b]The activations contain both input and output fmaps of a layer.

inserted in buffer allocation and scheduling. Specifically, input x is first allocated in IB and then loaded from EM to IB for CONV1. After completing the CONV1 operation, a check is made to determine if the IB space for x is still sufficient to process CONV2 without performance degradation during the optimization of the buffer allocation. If the space is sufficient, the buffer for the three components is allocated with the remaining buffer space, except for the x space for CONV3. Otherwise, four data components including x are used for the buffer allocation. For searching the scheduling spaces for CONV3, the input component for x is added as a parameter. When x is stored in the IB during the processing of CONV2, the data-reuse parameter for x is set, and because x is not moved from EM to IB (ie, x data reuse), double buffering for x is not used.

5 | EVALUATION

We evaluate the proposed architecture and optimizations in this section. The experimental setup is first described and then the experimental results are presented.

5.1 | Experimental environment

We have presented the accelerator model on the basis of the architecture presented in [24] as the baseline. However, the baseline architecture was modified according to the proposed buffer, datapath, and data control in Section. 3. In addition, we designed a simulator engine to evaluate the configurable architecture. The configurable architectural parameters are as follows: the PE array size, buffer sub-block size, number of sub-blocks for a buffer row, and diagonal-path availability. For the complete accelerator system, we implemented the system model using DDR4 memory for EM and 256-bit-width advanced extensible interface system bus for the data movement between EM and IB. For an EM-access module, two read channels and one write channel, which are responsible for the input, output, and weight components, respectively, were additionally designed for the experiments.

We evaluate the proposed architecture and optimization algorithms using the popular CNNs summarized in Table 1. This is because we target the high-performance accelerator implementation and CNN process operations with higher complexity, resulting in more complicated dataflow control and a larger activation and weight size than those of MLPs and RNNs. Notably, AlexNet [19], VGGNet [20], and ResNet [18] are well-known NNs that have won in ImageNet ILSVRC [25] for image classification. Yolov2 [21] and yolov3 [22] were developed to detect objects in real-time so that they could be used in many practical applications. SqueezeNet [23] was designed for lightweight applications with small parameters. The test NNs have different layer structures, memory footprints, and number of operations.

5.2 | Hardware-configuration comparison

In Figure 9, we compare several NNs based on the speedup of the processing time for various hardware architecture configuration factors. All the optimization algorithms mentioned in Section 4 are applied to ensure a fair comparison. The baseline architecture is configured using a 128×128 PE array and 256 KB of a memory row with 8 sub-blocks. Each configuration of the baseline is changed while maintaining other configurations, and the hardware performance with the changed configuration is calculated and compared with the baseline configuration. The speedup for comparison between the baseline and the changed configurations is computed as follows:

$$\text{Speedup} = \frac{\text{Processing Time}_{\text{baseline}}}{\text{Processing Time}_{\text{changed}}} \tag{2}$$

where the processing time for the target NN is estimated by the simulator. Compared with SqueezeNet, the performance of VGG-16 is dramatically affected by the PE array size and buffer size because of the large data size, including weights and fmap. However, the number of sub-blocks is less impactful because the optimization compensates for the degradation by

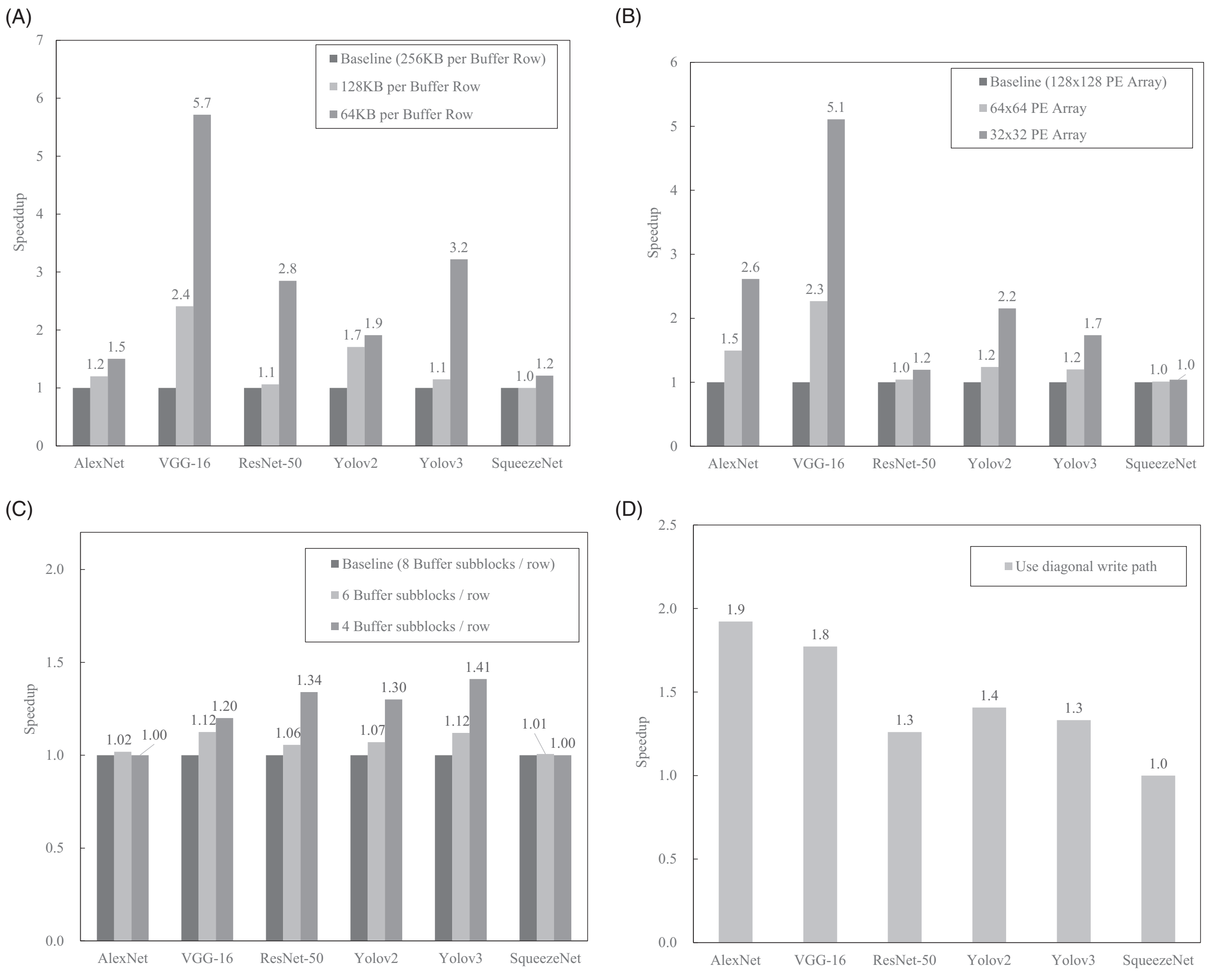


FIGURE 9 Performance comparison on the basis of hardware configurations: (A) buffer size, (B) PE array size, (C) number of sub-blocks, and (D) use of the diagonal write path

decreasing the number of sub-blocks. On the other hand, the running time for the optimization with a small number of sub-blocks is increased because of the inferior hardware condition. The diagonal write path improves the performance by 1.44× on average, as it decreases the data movement between IB and EM to rearrange the data. The performance of SqueezeNet is maintained in almost all the configurations, as it is too small to affect

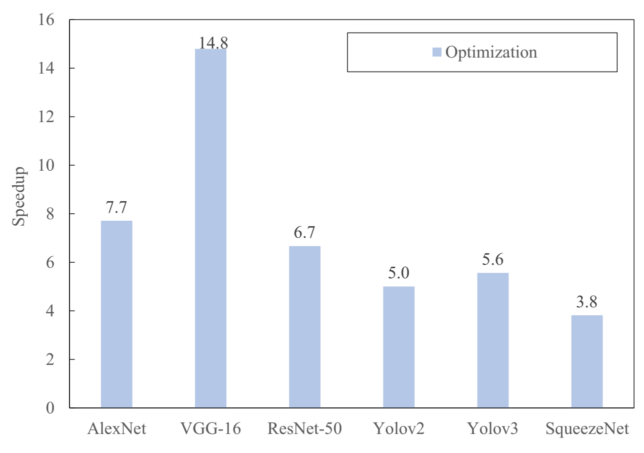


FIGURE 10 Efficiency of automated optimization algorithm

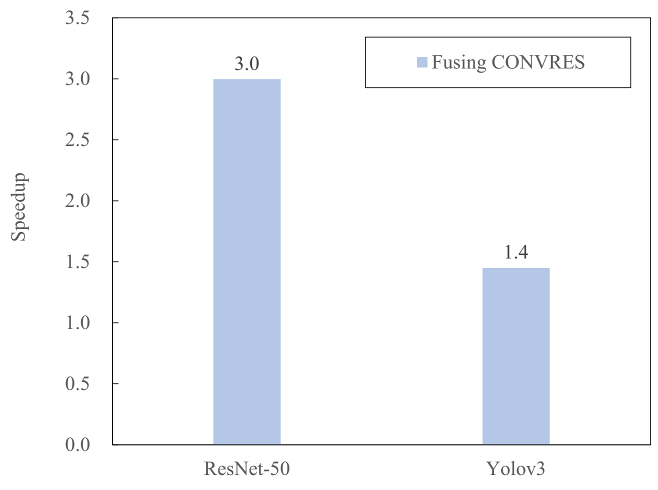


FIGURE 11 Efficiency of fusing CONV and RES

the hardware configurations that provide high performance. Moreover, SqueezeNet has been fully optimized using the optimization algorithm by maximizing data reuse in IB.

5.3 | Optimization performance

The efficiency of the optimization algorithms is evaluated by comparing their performances with that of the baseline. To that end, we exploited the baseline hardware mentioned in Section 5.2. The baseline optimization is defined using single buffering for all the data components and unified in/out buffer. For the unified buffer, we generated the special PE array control command to prevent the SRAM read and write conflict by using a delay. The unified buffer option degrades the performance of the PE operation because of the delay cycles while increasing the data reuse in IB via reduced fragmentation. Therefore, the automated optimization system enhances the performance by analyzing the layer parameters of NN and searching the best optimization condition. The speedup comparison between the baseline and the optimized algorithms is depicted in Figure 10. Consequently, the performance is considerably improved in the case of the following automated optimization algorithms: buffer-allocation algorithm (including slicing, tiling, and assigning flexible buffer) and scheduling-search algorithm. Expectedly, the automated optimization algorithm significantly improves the performance.

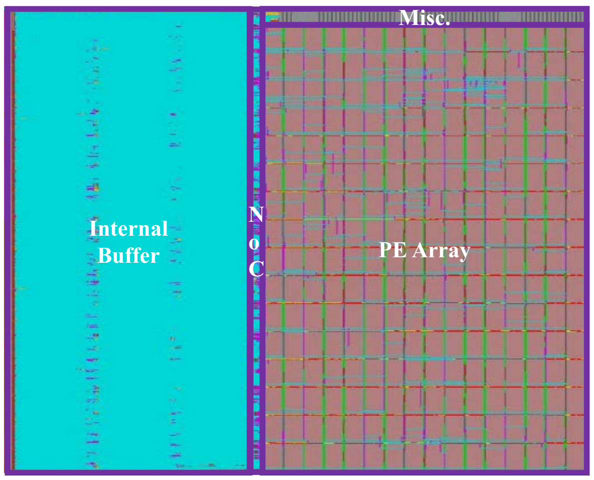
5.4 | Analysis of fusing operations

To analyze the efficiency of the fusing operations, we consider CONVRES of Yolov3 and ResNet as the test layer. Because the evaluated NNs include 23 and 16 RES operations, respectively, as presented in Table 1, the performance improvement due to the fusing operations significantly affects the overall performance. In Figure 11, we depict the performance comparison between the fused (ie, CONVRES) operation and separate operations (ie, CONV and RES, respectively). The speedups of ResNet and Yolov3 are 3× and 1.4×, respectively, as the fusing technique minimizes the data movement between IB and EM or between PEs and IB. Specifically, our output-stationary based architecture is considerably effective when fusing and processing the operations without weights such as RES and POOL. In conclusion, high-performance accelerators involve operation-fusing optimization.

5.5 | Hardware implementation

The designed accelerator was synthesized using the Synopsys Design Compiler with the TSMC 28-nm library, and it achieved an operating frequency of 1 GHz. The layout of the synthesized architecture with a 128 × 128 PE array is depicted in Figure 12A. A PE is designed for float16 operators and 32 TFLOPS is supported. The IB comprises 128 rows to fit to a PE array size, and a buffer row is divided into 8 sub-blocks where a buffer sub-block is 32 KB. Consequently, the accelerator is implemented using a 32 MB on-chip SRAM and 16 384 PEs. The chip-area estimate is 494 mm^2 from the layout and the area breakdown of the synthesized accelerator is depicted in Figure 12B. NoC accounts for only 0.23%, as we implement the policy that NoC is connected only between adjacent IB rows and PEs and between PEs and PEs. The miscellaneous includes the data-movement-management unit between IB and EM, DAC, and the main control logics. Therefore, the high-performance accelerator, which can operate at high frequency, can be achieved using a small programmable dataflow and data-movement controller, when the PE-array and buffer sizes are largely increased because of the systolic-array-based architecture and simple NoC structure. We additionally estimate the power consumption of the

(A)



(B)

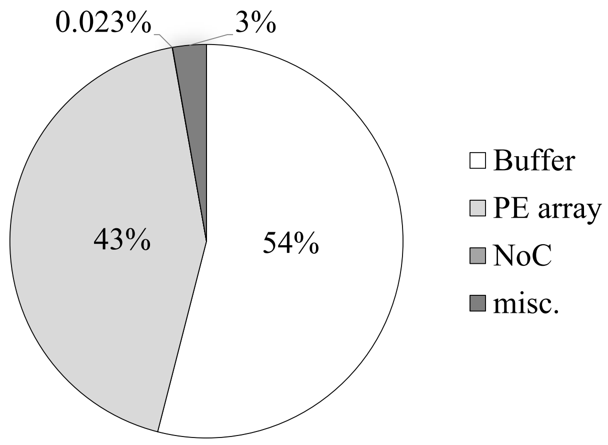


FIGURE 12 Synthesized result of the proposed DNN accelerator: (A) layout of the designed DNN accelerator and (B) area breakdown of the synthesized architecture

synthesized accelerator using Synopsys Prime Time. The static power dissipates by approximately 15 W and a 12.5 W dynamic power consumption is estimated by applying a 5% toggle rate.

6 | CONCLUSIONS

We presented a flexible buffer structure, effective dataflow for fusing operations, and programmable data-access control for high-performance DNN accelerators. Based on the proposed architecture, we automated the optimization algorithms to maximize performance. The parameters and search spaces were defined, and the automation algorithm based on the exact hardware model was proposed to efficiently operate the hardware. These combined techniques achieved the implementation of a high-performance accelerator with a large PE array and IB operating at a high frequency, thereby providing the convenience for processing various NNs. In addition, the results of the fusing operations showed the possibility of tighter optimizations. However, automating the operation fusion remains to be explored in the future.

ORCID

HyunMi Kim https://orcid.org/0000-0003-4105-7639

REFERENCES

1. Y. LeCun, Y. Bengio, and G. Hinton, *Deep learning*, Nature **521** (2015), 436–444.
2. L. Besacier et al., *Automatic speech recognition for under-resourced languages: a survey*, Speech Commun. **56** (2014), 85–100.
3. K. Arulkumaran, A. Cully, and J. Togelius, *AlphaStar: An evolutionary computation perspective*, arXiv preprint arXiv:1902.01724v2, 2019.
4. M. Fatima and M. Pasha, *Survey of machine learning algorithms for disease diagnostic*, J. Intell. Learn. Syst. Aapplicat. **9** (2017), 1–16.
5. S. Grigorescu et al., *A survey of deep learning techniques for autonomous driving*, arXiv preprint arXiv:1910.07738, 2019.
6. I. S. Krizhevsky and G. E. Hinton, *ImageNet classification with deep convolutional neural networks*, in Proc. Int. Conf. Neural Inf. Process. Syst. (Nevada, USA), 2012, 1097–1105.
7. J. Albericio et al., *Cnvlutin: Ineffectual-neuron-free deep neural network computing*, in Proc. Int. Symp. Comput. Architecture (Seoul, Rep. of Korea), (2016), 1–13.
8. S. Han et al., *EIE: Efficient inference engine on compressed deep neural network*, in Proc. Int. Symp. Computer Architecture (Seoul, Rep. of Korea), (2016), 243–254.
9. Y.-H. Chen et al., *Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks*, IEEE J. Solid-State Circuits **52** (2017), 127–138.
10. Y. Chen et al., *DaDianNao: A machine-learning supercomputer*, in Proc. Int. Symp. Microarchitecture (Cambridge, UK), (2014), 609–622.
11. N. Jouppi et al., *In-datacenter performance analysis of a tensor processing unit*, in Proc. Int. Symp. Computer Architecture (Toronto, Canada), (2017), 1–12.
12. Y. H. Chen et al., *Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices*, IEEE J. Emerg. Sel. Top. Circuits Syst. **9** (2019), 292–308.
13. V. Sze et al., *Efficient processing of deep neural networks: A tutorial and survey*, Proc. IEEE **105** (2017), 2295–2329.
14. R. Andri et al., *YodaNN: An architecture for ultra-low power binary-weight cnn acceleration*, IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **37** (2018), 48–60.
15. Y. C. Yoon et al., *Image classification and captioning model considering a CAM-based disagreement loss*, ETRI J. **42** (2020), 67–77.
16. J. Jung and J. Park, *Improving visual relationship detection using linguistic and spatial cues*, ETRI J. **42** (2020), 399–410.
17. J. A. B. Fortes and B. W. Benjamin, *Systolic arrays - From concept to implementation*, IEEE Comput **20** (1987), 12–17.
18. K. He et al., *Deep residual learning for image recognition*, in Proc. IEEE Conf. Comput. Vision Pattern Recogn. (Nevada, USA), 2016, 770–778.
19. I. Sutskever Krizhevsky and G. Hinton. *Imagenet classification with deep convolutional neural networks*, in Proc. Adv. Neural Inf. Process. Syst. (Nevada, USA), 2012, 1106–1114.
20. K. Simonyan and A. Zisserman, *Very deep convolutional networks for large-scale image recognition*, in Proc. Int. Conf. Learn. Representations (San Diego, USA), 2015.
21. J. Redmon and A. Farhadi, *YOLO9000: Better, faster, stronger,* arXiv preprint, arXiv1612.08242, 2016.
22. J. Redmon and A. Farhadi, *Yolov3: An incremental improvement,* arXiv preprint, arXiv:1804.02767, 2018.
23. F. N. Iandola et al., *Squeezenet: Alexnet-level accuracy with 50x fewer parameters and; 0.5 mb model size*, arXiv preprint, arXiv:1602.07360, 2016.
24. Y. Kwon et al., *Function-safe vehicular ai processor with nano core-in-memory architecture*, in Proc. IEEE Int. Conf. Art. Intel. Circuits Syst. (Hsinchu, Taiwan), 2019, 127–131.
25. O. Russakovsky et al., *ImageNet large scale visual recognition challenge*, arXiv preprint, arXiv:1409.0575, 2014.

AUTHOR BIOGRAPHIES

HyunMi Kim received her BS and MS degrees in Electronic Engineering from Inha University, Incheon, Rep. of Korea in 2004 and 2006, respectively, and her PhD degree in Computer Software from the University of Science and Technology, Daejeon, Rep. of Korea in 2018. Since 2012, she has been with the Electronics and Telecommunications Research Institute and is currently with the AI SoC Research Department as a senior engineer. Her research interests are in neural network systems including AI processors and DL compilers, SoC architecture design, optimization algorithms for SoC systems, and signal processing for multimedia applications.

Chun-Gi Lyuh received his BS degree in Computer Engineering from Kyungpook National University, Daegu, Rep. of Korea in 1998. He received his MS and PhD degrees in Electrical Engineering and Computer Science from the Korea Advanced Institute of Science and Technology (KAIST), Daejeon, Rep. of Korea in 2000 and 2004, respectively. He joined the Electronics and Telecommunications Research Institute (ETRI), Daejeon, Rep. of Korea in 2004 and is currently a principal member of the research staff. His current research interests include deep learning processors for mobile hardware and software development kits.

Youngsu Kwon received his BS, MS, and PhD degrees from the Korea Advanced Institute of Science and Technology (KAIST), Rep. of Korea in 1997, 1999, and 2004, respectively. He was a Postdoctoral Associate at the Microsystems Technology Laboratory (MTL), Massachusetts Institute of Technology from 2004 to 2005, designing 3-dimensional FPGA. He has been with the AI SoC Research Department, Electronics and Telecommunications Research Institute (ETRI), Rep. of Korea since 2005. At ETRI, he is the Director and Principal Researcher of the AI SoC Research Department that is devoted to the design of the AI processor, AB. He has special interests in many-core architecture, AI processor design, low-power architecture design, computer-aided design, and algorithmic optimizations of circuits and systems. He received the Presidential Prize from the Korean Government in 2016, Official Commendations from the Ministry of Science and ICT as well as the Ministry of Industry in 2016, the Excellent Researcher Award from the Korea Research Council in 2013, the Industrial Contributor Award from the Korean Federation of SMEs in 2013, and medals from Samsung's Thesis Prize in 1997 and 1999.