Received 26 July 2020; revised 7 October 2020; accepted 28 October 2020. Date of publication 9 November 2020; date of current version 25 January 2021.

*Digital Object Identifier 10.1109/OJCAS.2020.3035402*

# MOSDA: On-Chip Memory Optimized Sparse Deep Neural Network Accelerator With Efficient Index Matching

**HONGJIE XU (Student Member, IEEE), JUN SHIOMI (Member, IEEE), AND HIDETOSHI ONODERA (Fellow, IEEE)**

Department of Communications and Computer Engineering, Kyoto University, Kyoto 606-8501, Japan

This article was recommended by Associate Editor X. Zhang.

CORRESPONDING AUTHOR: H. XU (e-mail: xuhongjie@vlsi.kuee.kyoto-u.ac.jp)

This work was supported in part by the Doctoral Program for World-Leading Innovative and Smart Education, Ministry of Education, Culture, Sports, Science and Technology and in part by the VLSI Design and Education Center, the University of Tokyo in collaboration with Cadence Design Systems, Inc., Synopsys, Inc., and Mentor Graphics, Inc.

**ABSTRACT** The irregular data access pattern caused by sparsity brings great challenges to efficient processing accelerators. Focusing on the index-matching property in DNN, this article aims to decompose sparse DNN processing into easy-to-handle processing tasks to maintain the utilization of processing elements. According to the proposed sparse processing dataflow, this article proposes an efficient general-purpose hardware accelerator called MOSDA, which can be effectively applied for operations of convolutional layers, fully-connected layers, and matrix multiplications. Compared to the state-of-art CNN accelerators, MOSDA achieves 1.1× better throughput and 2.1× better energy efficiency than Eyeriss v2 in sparse Alexnet in our case study.

**INDEX TERMS** Sparsity, DNN, machine learning, hardware accelerator.

## I. INTRODUCTION

In recent years, the prediction accuracy of Deep Neural Network (DNN) has been steadily increasing with the improvement of network structure [1]. As a result, DNN has been widely used in all aspects of life such as image processing and language modeling. Therefore, many endpoint terminals integrate Application-Specific Integrated Circuits (ASICs) to perform local real-time inference [2]–[4]. However, high-precision prediction is often accompanied by an increase in computational complexity [5], [6].

As DNN networks become deeper and more complex, the required computing power and energy consumption are also increasing [7]–[9]. Since most endpoint devices are battery-powered, energy-efficient ASICs which are able to process DNN is highly required. The matrix operation, which consists of convolution (CONV) operations and matrix multiplications, consumes more than 90% of the energy in DNN [8]. Therefore, many studies have focused on designing low-energy but high-throughput accelerators for matrix operations [10]–[18]. Reference [8] proposes a key property called *data reuse* to describe the number of accesses to the same data during the processing. Exploiting data reuse has achieved great success to reduce expensive memory accesses in DNN processing. Previous works [11]–[13] exploit data reuse in the dataflow which describes the sequence of data movement in the memory hierarchy and the sequence of operation executions in arithmetic logic units (ALUs).

In order to adopt limited computing resources of endpoint devices, many studies utilize pruning techniques to reduce the total amount of parameters in DNNs [19]–[23]. Since DNN processing often uses Rectified Linear Unit (ReLU), this means that many pixels in feature maps are also be zeroed. As a result, the matrices involved in DNN processing are often sparse [24]. Focusing on the sparsity of DNN processing, previous hardware accelerators transmit only non-zero data to the processing elements (PEs), thereby translating the sparsity into energy reduction of the data movement [13], [25], [26]. Once non-zero data are loaded into PEs, hardware implementations are expected to fulfill data into all available ALUs by index matching [18]. Index

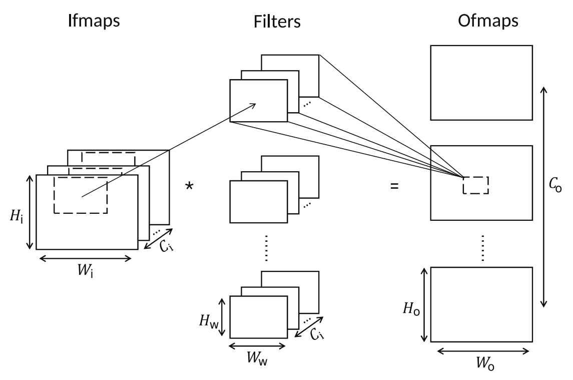

FIGURE 1. High dimensional convolutions in dense DNN: $N \times C_i \times C_o \times H_o \times W_o \times H_w \times W_w$ multiplication operations are required in total.

matching is to match the coordinate of the weight and the coordinate of the input feature map (ifmap) pixel for each operation to determine whether there is a meaningful operation that produces a non-zero product term. However, the irregular distribution of non-zero data leads to a large matching overhead in parallel processing since accelerators have to match multiple coordinates in parallel [9].

Index matching does not affect parallel-processing performance in dense DNN, since all processing dependencies between ifmaps and weights are clear at the design time. In sparse DNN, index matching becomes critical to decide the parallel-processing hardware complexity due to the data irregularity. The index matching comes to be complicated in the parallel processing task, which should check all loaded weight coordinates and ifmap pixel coordinates with each other to find out meaningful operations that lead to non-zero products. For example, Eyeriss v2 [13] enhances throughput by utilizing multiple scratchpad memories with a pipeline architecture inside PEs to detect meaningful operations. Fetching just right amount of input data with a simple logic to guarantee high PE utilization is important, since the number of non-zero products is unknown in sparse DNN processing. In order to detect meaningful operations quickly in sparse DNN processing, it is required to find out the property to help us understand how many meaningful operations are among input feature map (ifmap) pixels and weights, thereby guiding us to design a simple sparse processing architecture. We view meaningful operations as multiplications that both corresponding ifmap pixels and weights are non-zeros.

To address the challenge of high PE utilization, this article makes the following contributions:

1) This article first describes the property called matching type for index matching in DNN processing, which helps to predict the number of meaningful operations efficiently, thereby enhancing PE utilization in sparse DNN processing. We utilize PE utilization to refer to the average ratio of multipliers that process meaningful operations in the PE.
2) Based on the proposed property, this article first introduces a sparse DNN accelerator called Memory Optimized Sparse DNN Accelerator (MOSDA). This article makes a hardware evaluation of the proposed sparse accelerator to compare with state-of-the-art DNN accelerators in both dense and sparse DNN processing.

TABLE 1. DNN shapes.

| Parameters | Description |
|---|---|
| $N$ | Ifmap Batch Size |
| $H_i/W_i$ | Ifmap Height / Width |
| $H_w/W_w$ | Weight Height / Width |
| $H_o/W_o$ | Ofmap Height / Width |
| $C_i$ | Ifmap Channel Size |
| $C_o$ | Ofmap Channel Size |
| $D$ | Depthwise Channel Size |
| $U$ | Stride Size |

The rest of this article is organized as follows. Section II discusses properties in DNN processing. Section III introduces the proposed DNN processing dataflow which is able to process both dense and sparse DNNs. Section IV introduces the hardware realization of the proposed processing dataflow. Section V discusses the simulation results to show the superiority of the proposed hardware realization. Section VI concludes this article.

## II. BACKGROUND

### A. DNN PROCESSING

DNN generates classification results by performing feature extraction from raw input data. DNN improves model accuracy by very deep hierarchy of layers in the network. CNN is commonly applied to analyzing image recognitions. As primary layers in CNN, CONV layers use a group of filters to extract high-level features which are called output feature maps (ofmaps) from ifmaps. As shown in Fig. 1, traditional CNNs often have multi-channel 2-dimensional (2D) ifmaps and multi-channel 2D weights to enhance prediction accuracy [27]. Given the convolution shapes in Tab. 1, the CONV operation with multiple batches can be expressed as:

$$\begin{aligned}\mathbf{O}[z][u][x][y] &= \mathbf{B}[u] + \sum_{k=1}^{C_i}\sum_{i=1}^{W_w}\sum_{j=1}^{H_w}\mathbf{I}[z][k][Ux+i][Uy+j] \\ &\quad\times \mathbf{W}[u][k][i][j], \\ &\quad 1 \le z \le N,\ 1 \le u \le C_o,\ 1 \le x \le W_o \\ &\quad 1 \le y \le H_o, \\ H_o &= (H_i - H_w + U)/U \\ W_o &= (W_i - W_w + U)/U, \qquad (1)\end{aligned}$$

where **O**, **B**, **W** and **I** indicate the matrices of ofmaps, biases, weights and ifmaps, respectively [8].

Other primitive operations in DNN can also be represented by Eq. (1). In order to convert the high-level extraction features from CONV layers into the classification results, Fully-Connected (FC) layer is used between CONV layer and the output layer [1]. Recent CNN has developed depthwise separable convolution [6] to diminish CONV channels to save energy. Pointwise convolution acts as the operations

**TABLE 2.** Relations between CONV and other primitive operations in DNN.

| Processing | Description |
|---|---|
| Pointwise Convolution | $H_w = W_w = 1$ with Eq. (1) |
| Depthwise convolution | Eq. (2) |
| FC (Matrix Multiplication) | $H_i = W_i = H_w = W_w = 1$ with Eq. (1) |

with $H_w = W_w = 1$ with Eq. (1). In mathematics, the operations in FC layers and Multilayer Perceptron (MLP) [1] can be represented by Eq. (1) with $H_i = W_i = H_w = W_w = 1$. The depthwise convolution consists of the 2D CONV processing with multiple ifmap channels and the same number of ofmap channels. In this case, the accumulation pattern is different from naive convolution operations. We thus need to utilize another dimension to describe the channel in depthwise convolution which will be represented by the following equation.

$$
\begin{aligned}
\mathbf{O}[z][d][x][y] &= \mathbf{B}[z][d] + \sum_{i=1}^{W_w}\sum_{j=1}^{H_w} \mathbf{I}[z][d][Ux+i][Uy+j] \\
&\quad \times \mathbf{W}[d][i][j] \\
&1 \le z \le N,\ 1 \le d \le D,\ 1 \le x \le W_o \\
&1 \le y \le H_o \\
H_o &= (H_i - H_w + U)/U \\
W_o &= (W_i - W_w + U)/U \qquad (2)
\end{aligned}
$$

The relationship is summarized in Tab. 2. Since stride $U$ does not affect the key characteristics of convolution, we default all $U$ in DNN processing to 1 in the rest of this article.

The sparsity of weights and ifmaps can be utilized by gating or skipping operations, thereby improving energy efficiency. In order to further enhance throughput by sparsity, a complex control logic is usually inevitable according to the irregularity of weights and ifmaps in sparse DNN. For achieving high PE utilization, we will propose a method that aims to directly infer the number of operations from the number of loaded non-zero ifmap pixels and the number of loaded non-zero weights so that we can easily arrange weights and ifmap pixels required for massively parallel processing. We fetch just the right amount of input data, thereby maximizing PE utilization with a simple control logic. We focus on the data dependency between ifmaps and weights to evaluate its impact on the number of required operations for sparse DNN.

## B. MOTIVATIONAL EXAMPLE

In this section, we review a sparse matrix multiplication shown in Fig. 2. The shape of the matrix multiplication is $C_i = C_o = 2$. Suppose we have only 2 multipliers available, two possible processing dataflows, Dataflow A and Dataflow B, are shown in Figs. 2 (A) and (B), respectively. Dataflow A uses 2 multipliers to generate product terms to the same ofmap pixel in each step. Dataflow B uses 2 multipliers to generate product terms from the same ifmap

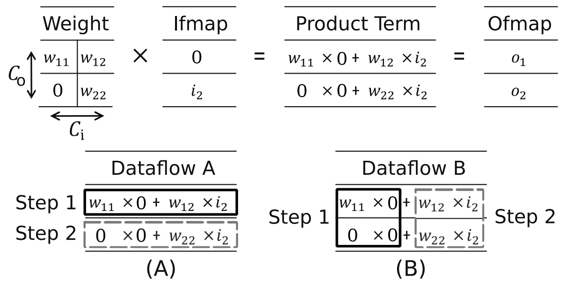

**FIGURE 2.** Sparse Matrix Multiplication with $C_i = C_o = 2$, $N = H_w = W_w = H_i = W_i = H_o = W_o = 1$. Two possible processing dataflows with 2 multipliers are introduced.

pixel in each step. According to the definition given in [28], Dataflow A is an ifmap-stationary dataflow. Dataflow B is an ofmap-stationary dataflow. Both dataflows require two steps to complete the processing when matrices are dense.

For a sparse matrix operation shown in Fig. 2, all required data have been loaded into the on-chip buffer, and each buffer can only hold one data for initialization. We observe that the number of multiplications processed in a single step in different dataflows is not the same.

Dataflow A performs two multiplications which have the same $C_o$ coordinate in parallel. In the first step, one multiplication is required since the non-zero ifmap pixel $i_2$ should multiply with one non-zero weight $w_{12}$, while there are two non-zero weights available. In the second step, one multiplication is required since the non-zero ifmap pixel $i_2$ should multiply with one non-zero weight $w_{22}$. In Dataflow A, it is difficult to infer the number of multiplications required until we fetch non-zero data into multipliers. It is hard to fulfill all multipliers without additional control logic in Dataflow A.

Dataflow B performs two multiplications which share the $C_i$ coordinate. Since $i_1$ is zero, there is no required operation in the first step of dataflow B. As a result, the first step can be skipped once we notice $i_1$ is zero. In the second step, two multiplications between one ifmap pixel $i_2$ and two weights $w_{12}$, $w_{22}$ are processed.

In Dataflow B, we can easily infer the number of operations required by the non-zero ifmap amount times non-zero weight amount. As a result, Dataflow B requires less hardware control effort to maximize PE utilization in our case study.

## C. SUBTASKS IN DNN PROCESSING

In order to compare different dataflows in sparse DNNs, we further clarify essential differences among processing dataflows. The number of operations for most existing DNN computing tasks ranges from millions [29] to billions [30]. It is difficult for acceleration hardware to realize fully parallel processing in such a scale. Therefore, we have to face a situation where the number of operations that can be processed within PEs is much smaller than the total number of operations. Under the situation of limited processing resources, it is required to decompose an entire processing task into *subtasks* to adapt the number of multipliers in a PE, which is

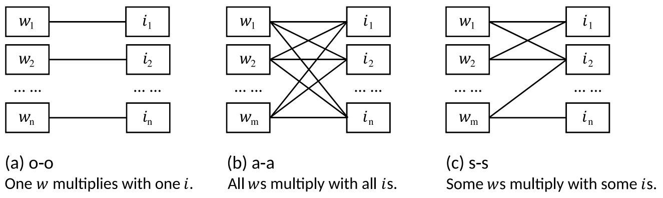
$w_1$
$w_2$
$w_n$
$w_m$
$i_1$
$i_2$
$i_n$
(a) o-o
One $w$ multiplies with one $i$.
(b) a-a
All $w$s multiply with all $i$s.
(c) s-s
Some $w$s multiply with some $i$s.

**FIGURE 3.** Three types of processing dependency. Type (a): one $w$ is processed with one $i$. Type (b): all $w$s are processed with all $i$. Type (c): The processing dependency between $w$s and $i$s is diverse.

**TABLE 3.** Matching types of typical subtasks in DNN.

| Subtask Processing Direction | Matching Type in Weights-to-Ifmap Pixels |
|---|---|
| $N$ | $[a-a]$ |
| $C_i$ | $[o-o]$ |
| $C_o$ | $[a-a]$ |
| $H_w$ | $[a-a]$ |
| $W_w$ | $[a-a]$ |
| $H_o$ | $[a-a]$ |
| $W_o$ | $[a-a]$ |
| $D$ | $[o-o]$ |
| $(H_o, H_w)$ | $[a-a]$ |
| $(C_i, C_o)$ | $[s-s]$ |

the same as *tiling level* in a PE defined in [31]. Each tiling level has a loop corresponding to each dimension in the original processing task. The tiling level in a PE contains the operation space and the associated dataspaces within registers in a PE [31]. One PE usually consists of multiple multipliers and registers. A subtask in a PE refers to the dimensions of processing that can be processed within data in one PE. Each subtask can be completed within a PE once data loaded into the registers inside. In DNN processing, a subtask refers to the processing task in the dimensions of $N$, $C_i$, $C_o$, $H_o$, $H_w$, $W_o$, $W_w$, or the multi dimensions such as $(H_o, H_w)$ and $(H_w, W_w)$.

Each subtask corresponds to a specific processing hardware implementation. It is the subtask in the processing dataflow that determines the hardware performance. Therefore, when we compare different processing dataflows, we should concentrate on the properties of the subtask in the PE.

## D. PROCESSING DEPENDENCY IN DNN

In dense DNN processing, it is simple to infer the number of operations required since the processing dependency between ifmaps and weights is clear at the design time. In sparse DNN, inferring the number of operations becomes critical for the processing speed and the hardware complexity. From the perspective of parallel processing hardware design, the processing dependency can be divided into three categories as shown in Fig. 3.

Type (a) $[o-o]$ in Fig. 3 refers that one ifmap pixel should be processed with only one weight in the processing task. If there are $x$ non-zero ifmap pixels and $y$ non-zero weights in sparse processing task, we can only infer that there are at most $x$ or $y$ operations required to be processed in the task. Therefore, a $[o-o]$ sparse processing task requires a complex hardware effort to maximize PE utilization.

Type (b) $[a-a]$ in Fig. 3 refers that all $n$ ifmap pixels should be processed with all $m$ weights in the processing task. If there are $x$ non-zero ifmap pixels and $y$ non-zero weights, we can easily infer that there are $xy$ operations required to be processed in the task. As a result, the number of operations can be easily inferred if the sparse processing task follows $[a-a]$ type.

Type (c) $[s-s]$ in Fig. 3 describes all the other cases of the processing dependency. Type $[s-s]$ refers that there are some ifmap pixels required to be multiplied with some weights in the processing task. The number of operations required for one ifmap pixel (or one weight) varies. As a result, in a sparse $[s-s]$ processing task, it is difficult to infer the overall required operations just by the number of non-zero data. Therefore, a $[s-s]$ sparse processing task requires dedicated hardware to maximize PE utilization.

We refer the processing dependency of weights and ifmap pixels in a certain processing task as *matching type*. In Dataflow A of Fig. 2, each ifmap pixel multiplies with only the matched weight in each subtask, which means that the matching type in the subtask of Dataflow A is $[o-o]$. In Dataflow B of Fig. 2, all ifmap pixels are multiplied with all weights in each subtask which means the matching type in the subtask of Dataflow B is $[a-a]$. Therefore, the hardware implementation of Dataflow A requires special hardware to maximize PE utilization without redundant data movement. On the other hand, the processing with [s-s] matching type can be found, for example, in the parallel processing with 4 multipliers. The parallel processing task of all four multiplications simultaneously in Fig. 2 belongs to the [s-s] matching type, which could lead to a complex index matching effort.

In $[a-a]$ matching type, the number of required operations required in the processing task can be simply expressed by the number of non-zero ifmap pixels times the number of non-zero weights regardless of sparsity. Therefore, we are able to fetch just the right amount of input data to enhance PE utilization with a simple control logic. As a result, keeping the matching type of the PE subtask being $[a-a]$ is the key to enhance PE utilization.

If we generalize the discussion from matrix multiplications to operations in DNN, matching types in different subtasks are summarized in Tab. 3. For convolution operations, the matching type of the corner ifmap pixels changes according to the chosen padding type. Since the central ifmap pixels usually dominate in number, we utilize matching type of the central ifmap pixels to represent the matching type of all ifmap pixels. Therefore, we view the matching type of convolution between ifmap pixels and weights as $[a-a]$. Available $[a-a]$ matching type refers to simple hardware workloads to process the subtask in parallel. In actual hardware accelerators, the chosen subtask for PEs is usually the combination of multi dimensions to maintain the throughput

**TABLE 4.** Processing dimension of subtasks in typical accelerators.

| Accelerator | Target | Subtask Processing Direction | Matching Type |
|---|---|---|---|
| Eyeriss v2 [13] | Sparse CNN | $(C_o, C_i, H_o, W_o, H_w, W_w, D)$ | $[s-s]$ |
| EIE [25] | Sparse FC | $(C_i, C_o)$ | $[s-s]$ |
| Cnvlutin [14] | Sparse CONV | $(C_i, C_o)$ | $[s-s]$ |
| OuterSPACE [18] | Sparse Matrix | $(C_o, H_o)$ | $[a-a]$ |

of the accelerators. In the case where the subtask contains both $[a-a]$ subtask and $[o-o]$ subtask, the matching type would be $[s-s]$. Once the subtask for DNN processing is determined, the matching type of the target subtask is fixed and will not change with sparsity.

### *E. RELATED WORK*

Tab. 4 introduces several typical state-of-the-art DNN accelerators. Different accelerators utilize different subtask processing strategies. This also leads to a difference in data-fetching strategies. The subtask adapted by Eyeriss v2 [13] loads data into registers in a PE as much as possible. Although this increases the theoretically available data reuse in PE subtask, it also makes its control logic complicated due to its $[s-s]$ matching type which requires a multi-stage pipeline architecture for each PE to enhance PE utilization. EIE [25], and Cnvlutin [14] have the processing dimension of $(C_i, C_o)$ in a PE, which leads to the $[s-s]$ matching type. OuterSPACE [18] utilizes outer product to replace the conventional inner product sequence to process matrix multiplications while keeping the matching type as $[a-a]$ in $(C_o, H_o)$ processing dimension. As a result, OuterSPACE achieves fast and efficient data scheduling. However, Outer SPACE can only handle matrix multiplication, but not other processing targets, such as convolution. In the next section, we propose a sparse DNN processing dataflow that fully considers the processing properties based on the dataflow in [32] derived for dense DNN processing. The proposed sparse processing dataflow shows efficiency when facing different processing targets (CONV, FC, and Matrix Multiplications) from our experiment result.

## III. DNN PROCESSING DATAFLOW

In this section, we propose a sparse DNN processing dataflow based on the matching type discussed in the previous section. The key idea of the proposed dataflow is to ensure that the matching type of the subtasks loaded into PEs is always $[a-a]$.

One batch of the convolution operation requires $C_i \times C_o \times H_o \times W_o \times H_w \times W_w$ times multiplications in total. Additions of the same order of magnitude are also required for multiplication results. As shown in Fig. 1, each ofmap pixel contains $C_i \times H_w \times W_w$ product terms, each product term needs to occupy one memory space in sequential processing. If we utilize fully parallel processing, we are able to merge $C_i \times H_w \times W_w$ product terms into one partial sum (psum) before accessing the memory to reduce the memory capacity

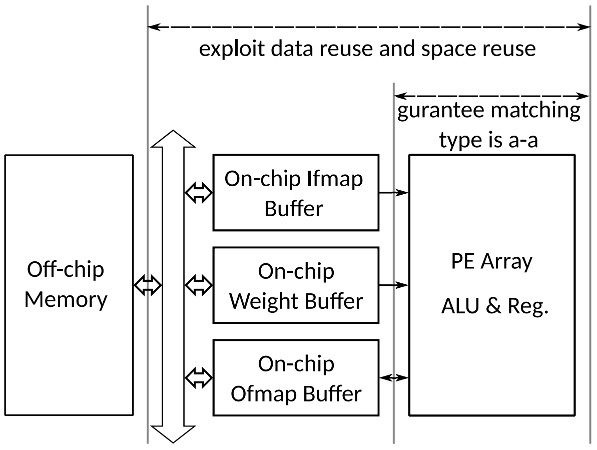

**FIGURE 4.** Requirements of sparse DNN processing dataflow. First, processing dataflow should guarantee the matching type of the subtask in the PE is $[a-a]$ for sparse DNN processing. Second, the dataflow should exploit data reuse and space reuse as much as possible.

required by the accelerator. The previous work refers to the processing property as *space reuse* [32]. Hence, this article notices that there are three processing properties that affect the hardware performance in DNN processing, which are the *data reuse*, the *space reuse* and the *matching type*.

In order to exploit three processing properties, as shown in Fig. 4, the sparse DNN processing dataflow follows two requirements. First, the processing dataflow for sparse DNN processing should guarantee all subtasks in the PE should be with the matching type $[a-a]$. Second, in order to further optimize the data movement in the hardware implementation for the dataflow, data reuse and space reuse should also be exploited as much as possible to reduce the number of memory accesses and memory capacity.

The dataflow in [32] for dense DNN processing satisfies both two requirements. In the dataflow, the subtask stored in the PE is 2D CONV, where matching type is $[a-a]$. In each clock cycle, the registers fetch multiple ifmap pixels in the same column to multiply with all weights in 2D CONV to exploit data reuse of ifmaps and weights. If we keep the subtask scope within 2D CONV, there are also product terms in the subtask that can be accumulated. In other words, there is also space reuse available in 2D CONV. Therefore, related product terms are added to reduce the required capacity of the on-chip memories.

This article first introduces a sparse DNN processing dataflow as an expansion of the dense dataflow in [32]. The proposed sparse processing dataflow does parallel processing within the $(C_o, H_o, W_o, H_w, W_w)$ dimensions to guarantee the matching type as $[a-a]$ in the PE. Under the target dimension, we further scale the processing range in the $(C_o, H_o, W_o)$ dimensions to match various sizes of processing tasks with a fixed number of multipliers inside a PE. The processing dataflow guarantees that the subtask in the PE is always $[a-a]$ matching type.

Fig. 5 introduces a processing example in one PE with 8 multipliers following the proposed processing dataflow. The

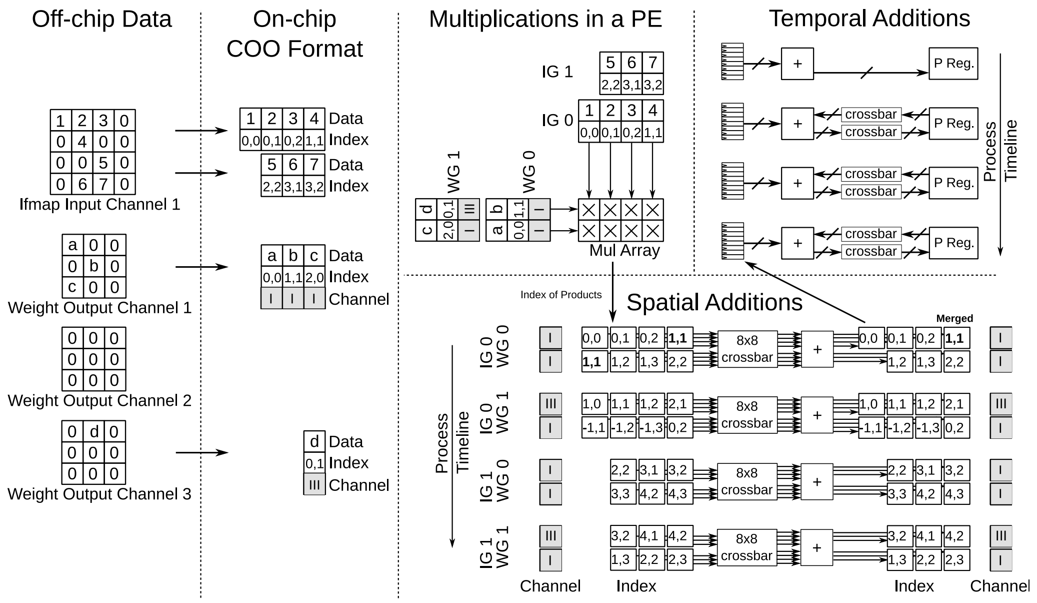

FIGURE 5. The subtasks loaded into the registers always belong to [*a* – *a*] matching type. Only non-zero ifmap pixels and weights are loaded into a PE.

target convolution consists of three 3 × 3 weight matrices for 3 different ofmap channels and a 4 × 4 ifmap matrix. Different ofmap channels contain different number of non-zero weights *a*, *b*, *c*, and *d*. First, all non-zero weights, ifmap pixels and the index coordinates in 2D CONV are stored in the COO format [33]. In addition, we store the ofmap channel coordinates I, II, and III in the weight register. The 8 multipliers are arranged as 2×4, where each set of 4 multipliers in the same row shares the same weight and each set of 2 multipliers in the same column shares the same ifmap pixel. Two weights are combined into one weight group (WG) and four ifmap pixels are combined into one ifmap group (IG). Once the data are loaded into registers, MOSDA fetches two weights in one WG and 4 non-zero ifmap pixels in one IG to fulfill all 8 multipliers per clock cycle. We adopt ifmap stationary here to exploit data reuse and space reuse. As shown in the right bottom of Fig. 5, (WG0 / IG0), (WG1 / IG0), (WG0 / IG1), and (WG1 / IG1) are processed sequentially.

When the product term is generated, we also compute the index information of the product term in the COO format. The ofmap channel of the product term is the same as the ofmap channel of the weight. Product terms in the same row share the same ofmap channel coordinate. We add product terms according to indexes of product terms in the spatial addition stage. By utilizing indexes of product terms, the spatial addition part shown in the right bottom of Fig. 5 merges product terms with the same indexes through a crossbar. For example, since there are two product terms that have the same index (1,1) in the first step (WG0 / IG0), two product terms are merged into one under the spatial addition stage. We store psum results from the spatial addition stage into registers and wait for the temporal addition. Psum results stored in registers are not always necessary for the

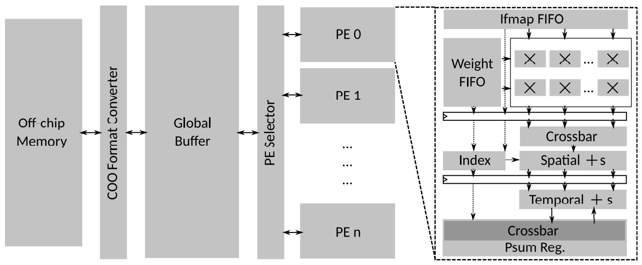

FIGURE 6. MOSDA architecture overview. The subtasks loaded into the registers always belong to [*a* – *a*] matching type. All PEs share one global buffer for all kinds of data.

following accumulation. For example, corner product terms with the index (−1, 1) in Fig. 5 is unnecessary because the index coordinates are out of the range of the output matrix. In order to avoid the energy loss by useless multiplications, the registers at the beginning of the temporal addition stage shield the useless product terms. The temporal addition part shown in the right above of Fig. 5 aims to further accumulate psums that can be accumulated between different clocks according to crossbars and the psum register.

Through the proposed dataflow, MOSDA can utilize the potential data reuse and space reuse of 2D CONV while ensuring the matching type as [$a-a$]. Therefore, the method can ensure the PE utilization while reducing the required memory access and memory capacity.

## IV. MOSDA ARCHITECTURE

### A. ARCHITECTURE OVERVIEW

A MOSDA architecture overview is shown in Fig. 6. All ifmap pixels and weights are stored in the off-chip memory. On-chip buffers store data which can be accessed in the near future to exploit data reuse and space reuse.

**TABLE 5.** **MOSDA design parameters.**

| MOSDA Parameter | Value |
|---|---|
| # PEs | 16 |
| Global Buffer | 192 KB |
| **PE Parameter** | **Value** |
| # Multipliers | 16 |
| Multiplier Width | 8 bits |
| Accumulation Width | 24 bits |
| Weight FIFO (per PE) | 32x16 bits |
| Ifmap FIFO (per PE) | 64x16 bits |
| Psum Register (per PE) | 64x32 bits |

MOSDA architecture can be scaled across a number of dimensions. Table 5 introduces key parameters of MOSDA in this article. MOSDA consists of 16 PEs, each with a 16 multiplier array. 16 multipliers in one PE can be arranged as 16 rows $\times$ 1 column for processing of FC layers or 2 rows $\times$ 8 columns for other cases. Each row shares the same weight, and each column shares the same ifmap pixel. We store ifmap pixels and weights in 8 bit. Product terms generated by multiplications are accumulated in 24 bit. When the accumulation is finished, 24-bit psums are converted back to 8-bit ofmap pixels and sent off-chip without coordinate information. We do not use the COO format to store data in the off-chip memory, since MOSDA is also intended for dense DNN processing. Memories carry an 8-bit overhead to encode the coordinate information for each value, which are able to process operations with 16 $\times$ 16 matrix in parallel. In the setup, if the sparsity is less than 50%, the data stored under the COO format will take more storage space than the data stored without the COO format. All data are transmitted in the COO format on-chip in this article.

## B. PARALLEL PROCESSING IN ONE PE

We exploit data reuse and space reuse, thereby reducing the cost of data movement within the PE. We utilize a three-stage pipeline architecture to reduce the critical path of the PE.

### 1) PARALLEL MULTIPLICATIONS

The main function of the first stage in the pipeline is parallel data fetching and multiplications. In order to exploit the data reuse and space reuse as much as possible, the dataflow further processes DNN ifmap channel by ifmap channel in the $C_\mathrm{i}$ dimension as an outer loop. In order to guarantee the matching type as [a-a], the dataflow processes the task ofmap channel by ofmap channel in the $C_\mathrm{o}$ dimension as an inner loop. If non-zero data inside one channel is not enough to fulfill multipliers, MOSDA will fetch data from multiple ofmap channels into one PE to maintain high PE utilization. Fig. 5 is a case study that loads data for 3 ofmap channels, since the number of operations is less than the number of multipliers within one ofmap channel. This ensures that PE utilization is not degraded because enough data is loaded into the PE.

For parallel processing of FC layers, 16 multipliers are configured into 16 $\times$ 1. All 16 multipliers share the same ifmap pixel, and each multiplier has a unique weight. For parallel processing of other cases such as convolutions, multipliers are configured into 2 $\times$ 8. 2 multipliers in one same column share the same ifmap pixel, and 8 multipliers in one same row share the same weight. Once multiplications are finished, MOSDA requires a crossbar logic to transmit psums from multipliers to spatial adders as shown in Fig. 6.

### 2) PARALLEL ADDITIONS

One of the key issues in sparse DNN processing architecture is high area/power overhead of accumulation due to irregular output data [34], [35]. In this article, we reduce the psum overhead by reducing psum register access by reserving a spatial redundant addition network. Fig. 6 introduces the psum accumulation logic in MOSDA, which is decomposed into a spatial addition stage and a temporal accumulation stage. The spatial addition stage aims to merge possible product terms into one psum within one clock cycle. We compare the indexes of all product terms with each other in the spatial addition stage, and merge corresponding product terms with the same index. The temporal accumulation stage further accumulates psums among different clock cycles. Both stages are controlled by coordinate information in index logics.

We merge the psums corresponding to the same ofmap pixel under the same step through a spatial addition network, thereby reducing the required psum register accesses. Due to the irregular output issue, a 16 $\times$ 16 crossbar is required between 16 product terms and 16 adders.

In the next temporal accumulation stage, we write merged psums into psum registers according to the coordinate index. Through the coordinate from the spatial addition stage, first we read the corresponding psums in the registers. Next, we add the read psums to the psums from the spatial addition stage and store them in psum registers. Once all the operations in the current ofmap channel have been completed, we write the psums back to the global buffer. Considering the worst case, we need to write 16 psums into 64 psum registers at the same time. Although this will cause a large area overhead, it ensures the MOSDA processing speed.

In order to limit register accesses by generated psums, we make the spatial addition and send psum results to psum registers. Considering the space reuse existed under different clock cycles, we continue to do the temporal accumulation for product terms stored in psum registers to reduce the required accesses to the global buffer.

## C. DATA TRANSMISSION AMONG PES

Due to the scale limitation of the all-to-all transmission network, the psum register capacity cannot be too large, thus limiting exploiting the data reuse in a single PE. We ensure that psum results of different PEs do not need to undergo further accumulation operations since they belong to different ofmap channels. The number of ALUs for a single PE is limited by its crossbar structure and cannot be large. MOSDA can increase the number of PEs for achieving higher throughput. As the number of PEs increases, throughput and

TABLE 6. MOSDA specifications.

| | |
|---|---|
| Technology | 65nm SOTB |
| Core Area | 2mm × 6mm |
| Gate Count (Logic Only) | 3066k gates (NAND-2) |
| On-chip Memory | 192 KB SRAM + 6 KB Register |
| Clock Frequency | 200 MHz |
| Peak Throughput | 153.6 GOPS |
| Filter Size | All |
| Ifmap Size | All |
| Processing Type | CONV, and FC (Matrix Multiplication) |

TABLE 7. Related energy consumption in convolution operations.

| Component | | Energy / Access [pJ] |
|---|---|---|
| 8-bit Multiplier ($E_{\mathrm{MUL}}$) | | 0.14 |
| 24-bit Adder ($E_{\mathrm{ADD}}$) | | 0.02 |
| 16×16 Crossbar ($E_{\mathrm{NoC}}$) | | 0.07 |
| 8-bit Register ($E_{\mathrm{REG}}$) | | 0.09* |
| On-chip SRAM ($E_{\mathrm{BUF}}$) | 8 kB | 3.31* |

*: Average energy consumption of read and write.

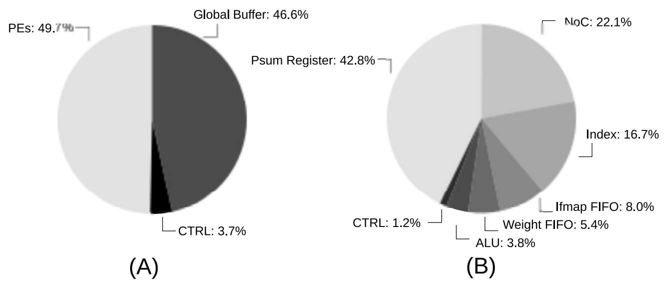

FIGURE 7. (A) MOSDA area breakdown. (B) Area breakdown in one PE.

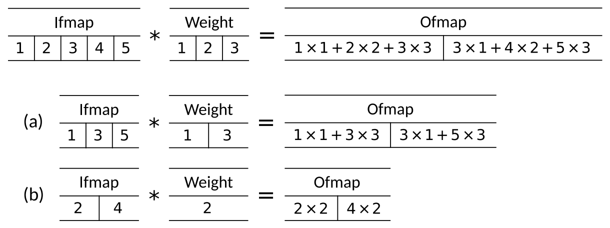

FIGURE 8. Decomposition method of 1-dimensional convolution when stride is larger than 1: In the case study, sizes of the ifmap pixel, the weight and the stride are 5, 3, and 2, respectively. When the stride is 2, the matching type will be [*s* – *s*]. Therefore, the convolution is decomposed into two smaller convolutions (a) and (b) with matching types almost [*a* – *a*].

power consumption will increase linearly. We use 16 PEs in this article as our case study. Different PEs are responsible for processing calculation tasks of different ofmap channels, thereby improving overall processing speed. Please note that a single PE may be responsible for multiple ofmap channels if the number of operations corresponding to a single ofmap channel is small compare to the number of multipliers in a PE, which ensures to maintain high PE utilization.

The global buffer provides the required non-zero weights and ifmap pixels to each PE sequentially. After data fetching finishes, PEs do processing in parallel. Once processing tasks in PEs are completed, each PE sequentially restores their respective psum results back to the global buffer.

MOSDA is able to switch the data transmission mode from sparse mode to dense mode to save energy overhead due to coordinate information. When MOSDA is initialized, information such as the size of the processing task and whether it is sparse is stored in the on-chip logic control. In the dense mode, MOSDA no longer activates the COO format converter or stores the coordinate information of ifmap pixels and weights in the global buffer.

## V. EXPERIMENTAL RESULTS

In this section, we verify the efficiency of MOSDA with 16 PEs by post-layout cycle-accurate gate-level simulations under a 65-nm Silicon-on-Thin-Box (SOTB) process with a 1.2 V supply voltage. The bandwidth between the global buffer and PEs transmit 48 words once for weights and ifmap pixels, or 24 words once for psums. The specifications are summarized in Tab. 6. The peak throughput of MOSDA is defined as the total throughput of processing elements, which consists of 256 multipliers and 512 adders in total. Tab. 7 summarizes the energy of major logic components in a 1.2 V supply voltage. All energy values except for on-chip SRAM are based on the average values for one operation during overall post-layout simulations. All hardware components are built in the fixed-point representation. More specifically, the energy cost of multipliers is collected by one multiplier, which has 8-bit inputs and one 16-bit output. The energy cost of adders is collected by one 16-bit adder for psum accumulations. The energy cost of the crossbars is evaluated by transmitting one 16-bit data through crossbars. The energy consumption of the access to the global buffer is collected from SRAM model in CACTI Ver. 7.0 [36] under a 65-nm process with a 1.2 V supply voltage.

MOSDA logic part is evaluated from the post-layout simulations. The data transmission Network-on-Chip (NoC) consists of transmission logics between weight FIFO and multipliers, logics between ifmap FIFO and multipliers, and a crossbar between multipliers and spatial adders. The area cost consumed by the crossbar is 85% of the total NoC area cost. The area of the psum register consists of both register cells and the crossbars to fetch data into registers. The energy and area of MOSDA buffer part are collected from CACTI model. Based on the code tracing logs, we calculate the access activities of the global buffer. Combining the activities and the physical SRAM data in Tab. 7, this article simulates the energy consumption of the overall MOSDA performance.

MOSDA core area is 12 mm$^2$, which consists of a 192 KB on-chip SRAM buffer and the 16 PEs. The overall logic gate count is 3066k NAND-2 gates. Fig. 7 (A) introduces the area breakdown in MOSDA. The global buffer accounts for 46.6% of the total core area. The control logic for data transmission between the global buffer and PEs consumes 3.7% of the total core area. The COO format convertor is included in the area breakdown of the control logic. The PE composed of multipliers and registers accounts for 49.7% of

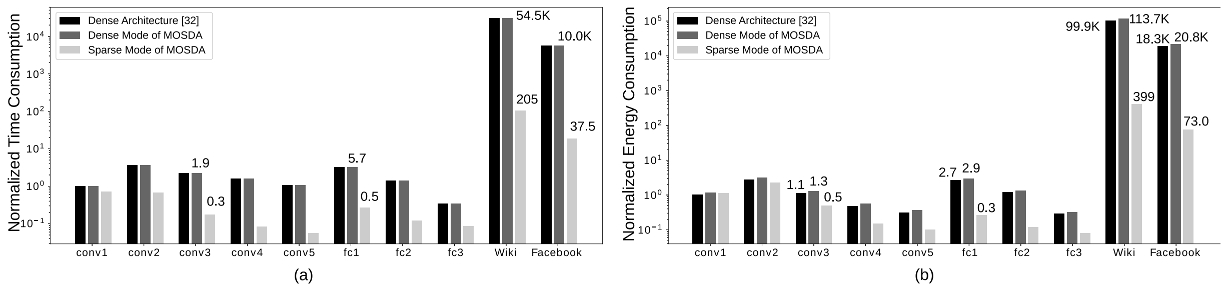

**FIGURE 9.** (a) Normalized time consumption in dense and sparse benchmarks including Alexnet and Matrix Multiplications. (b) Normalized energy consumption in dense and sparse benchmarks including Alexnet and Matrix Multiplications.

**TABLE 8.** Multiplier array utilization of MOSDA benchmarked with alexnet, and MobilNet that has batch size of 1 and matrix multiplications.

| Target | Non-zero Weight | Non-zero Weight Ratio | Non-zero Ifmap Ratio | Off-chip Memory Access [MB/feature] | Average Mul Utilization |
|---|---|---|---|---|---|
| CONV Layers 1-5 in Alexnet [37] | 2.2 M | - | - | 10.0 | 99.33% |
| FC Layers 1-3 in Alexnet [37] | 58.6 M | - | - | 58.7 | 99.84% |
| CONV Layers 1-5 in Sparse Alexnet [23] | 0.2 M | 7.7% | 69.8% | 4.3 | 90.01% |
| FC Layers 1-3 in Sparse Alexnet [23] | 5.9 M | 10.1% | 57.1% | 5.9 | 99.96% |
| CONV Layers 1-27 in MobileNet [6] | 4.1 MB | - | - | 5.0 | 99.05% |
| FC Layer 1 in MobileNet [6] | 1.0 MB | - | - | 1 | 97.71% |
| CONV Layers 1-27 in Sparse MobileNet [40] | 0.4 MB | 10.0% | 47.7% | 0.5 | 92.34% |
| FC Layer 1 in Sparse MobileNet [40] | 0.1 MB | 10.0% | 49.8% | 0.1 | 99.4% |
| Wiki-Vote [39] | 103.7 KB | 0.2% | 0.2% | 1.5 | 99.99% |
| Facebook [39] | 88.2 KB | 0.5% | 0.5% | 1.2 | 99.99% |

the total core area. Fig. 7 (B) introduces the reason for PE area overhead. In order to ensure all generated product-terms stored in registers in time, the psum register inside the PE requires 16 read-write ports, which consume 42.8% of the PE area. The accumulation overhead is the cost to handle the irregular output of sparse DNN processing. Simultaneously, the NoC (Crossbars) requires 22.1% of the PE area and the index logic requires 16.7%. Both parts require a complex data transmission logic to handle the unpredictable product terms generated from multipliers.

We simulate the energy consumption of MOSDA under various benchmarks including convolution layers, FC layers, and matrix multiplications. The benchmarks are introduced in Tab. 8, which consist of convolution layers and FC layers in Alexnet [37], sparse Alexnet [23], MobileNet [6] with width multiplier of 1.0 and input size of $224 \times 224$, corresponding sparse Mobilenet with data from ImageNet dataset [38], and two matrix multiplications by multiplying with itself in SNAP [39]. The stride $U$ of conv1 in Alexnet is 4. As a result, the matching type between weights and ifmap pixels in conv1 comes to be $[s-s]$ since not all weights are required to multiply with all ifmap pixels. To solve this issue, Fig. 8 introduces a decomposition method to convert the CONV into multiple smaller CONVs as shown in Figs. 8 (a) and (b). Small CONVs are all $[a-a]$ matching types (except the corner data). According to the decomposition method, we are able to process convolution layer with stride more than 1 with $[a-a]$ matching type.

Tab. 8 also introduces the average PE utilization in various benchmarks. PE utilization is the average ratio of multipliers that handle meaningful multiplications when PE is required to process operations. Note that the active rate of multipliers under data transmission time among memories is not taken into account. The key idea of MOSDA is to keep the matching type in the PE subtask always as $[a-a]$. The corner data in convolution processing does not have the matching type [a-a], which is the main reason for the PE utilization loss. As a result, the PE utilization of FC layer in Alexnet and matrix multiplications are as high as 99.96% and 99.99%, respectively. In CONV processing, although we assume that the matching type in convolution processing is $[a-a]$, the actual situation is that the corner data of the ifmaps in the convolution operation does not need to be multiplied by all weights. Therefore, this assumption led to a slight decrease of PE utilization in the sparse convolution processing, but still maintained at 90.01% and 92.34% for sparse Alexnet and sparse MobileNet, respectively.

Following the matching type of $[a-a]$, MOSDA keeps PEs always busy without redundant data fetching. Fig. 9 (a) shows the throughput comparison between MOSDA architecture from the dense mode to the sparse mode and a state-of-the-art dense processing architecture from Reference [32]. Benchmarks include Alexnet and Matrix Multiplications. The time consumption consists of processing time by multipliers, and data transmission time from the global buffer to registers in PEs. In order to make a fair comparison, both architectures consist of the same number of multipliers and the same on-chip bandwidth between the global buffer and PEs. Meanwhile, the sparse mode achieves $7.0\times$, $11.4\times$, and $265.9\times$ less time consumption than the dense mode in Alexnet conv3, Alexnet FC1 and Wiki-Vote Matrix Multiplication, respectively. The processing time is

**TABLE 9.** Comparison with the state-of-the-art DNN accelerators.

| | EIE [25] | OuterSPACE [18] | Eyeriss v2 [13] | This Work | | |
|---|---|---|---|---|---|---|
| Matching Type | $[s-s]$ | $[a-a]$ | $[s-s]$ | $[a-a]$ | | |
| Technology [nm] | 28 | 40 | 65 | 65 | | |
| Core Area | $63.8mm^2$ | $9mm^2$ | 2695k (NAND-2) | 3066k (NAND-2) | | |
| On-chip SRAM [KB] | 162 | 112 | 246 | 192 | | |
| Mul Num. | 256 | 32 | 384 | 256 | | |
| Core Frequency | 1200 MHz | 744 MHz | 200 MHz | 200 MHz | | |
| Bit Precision | 4 | 32 | 8 | 8 | | |
| Support FC Layer | Yes | Yes | Yes | Yes | | |
| Support CONV Layer | No | No | Yes | Yes | | |
| Sparse Model | | | Alexnet | Alexnet | ResNet-50 | MobileNet v2 |
| Throughput [Inference/sec] | - | - | 278.7 | 315.3 | 493.1 | 2764.4 |
| Energy Efficiency [Inference/J] | - | - | 664.6 | 1422.3 | 839.9 | 5574.6 |

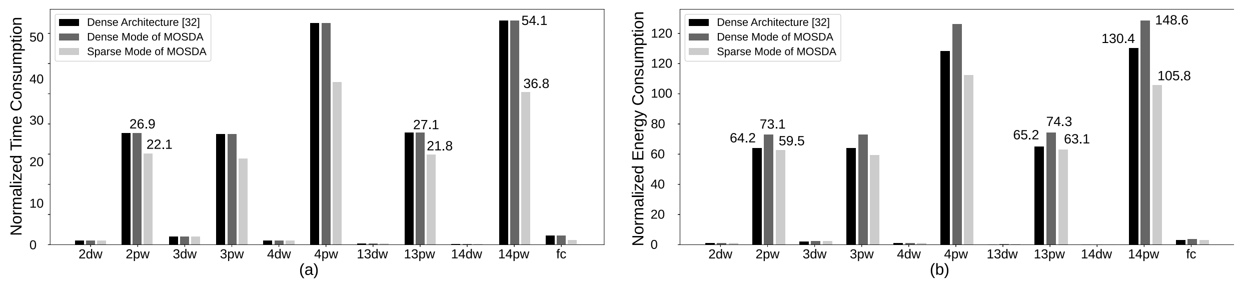

**FIGURE 10.** (a) Normalized time consumption in dense and sparse benchmarks including typical layers in MobileNet. (b) Normalized energy consumption in dense and sparse benchmarks including typical layers in MobileNet.

dramatically decreased with high PE utilization. Among all layers in dense Alexnet, data transmission between the global buffer and PEs consumes 38.1% of the on-chip time consumption. As we expected in sparse Alexnet, data transmission between the global buffer and PEs comes to be 62.2% of the on-chip time consumption, which infers that the time consumed by data transmission between the global buffer and PEs is hard to be reduced despite matching types inside the PE. The sparse mode achieves 5.3× better overall time reduction against the dense mode when processing Alexnet. The throughput of MOSDA dense mode is the same as the architecture in [32], since the number of multipliers and the on-chip bandwidth is set to be the same.

Fig. 9 (b) introduces the energy efficiency under various benchmarks. By considering sparsity, energy consumption is also reduced. The sparse mode achieves 2.6×, 11.2×, and 285.1× less energy consumption than the dense mode in Alexnet CONV3, Alexnet FC1, and Wiki-Vote Matrix Multiplication, respectively. Through the matching type $[a-a]$, MOSDA avoids redundant data movement from the global buffer to PEs. According to the post-layout simulation, the sparse mode's energy efficiency is 1507.8 inferences/J for sparse Alexnet. The sparse mode achieves 2.4× better overall energy reduction against the dense mode, which is 624.1 inferences/J. Fig. 11 shows the on-chip energy breakdown when processing all layers in sparse Alexnet. 42.6% of the energy is consumed in the access to the psum register due to its complex read-write logic. The global buffer consumes 30.9% of the energy. Crossbars to transmit data inside the PE lead to 9.3% energy overhead. The control logic including the COO format convertor consumes 3.7% of the overall energy consumption. Leakage in MOSDA consumes 1.0% of the overall energy consumption. As a cost to ensure $[a-a]$ matching type, reuse among ifmap channels is hard to be utilized by PEs, which brings an additional number of global buffer accesses. Comparing with the architecture in [32], MOSDA requires additional on-chip memory to store the coordinate information and the crossbar overhead for irregular data access pattern. Therefore, the dense mode of MOSDA suffers 1.4× larger energy overhead than the architecture in [32] in Alexnet.

Fig. 10 (a) shows MOSDA throughput enhancement from the dense mode to the sparse mode in several layers in MobileNet. The sparse mode achieves 1.2×, 1.2× and 1.5× less time consumption than the dense mode in MobileNet pointwise conv layer 2, pointwise conv layer 13, and pointwise conv layer 14, respectively. The data transmission time between the global buffer and PEs dominates the time consumption. Among all layers in dense MobileNet, data transmission between the global buffer and PEs consumes 83.0% of the on-chip time consumption. In sparse MobileNet, data transmission between the global buffer and PEs comes to be 95.5% of the on-chip time consumption. The sparse mode achieves 1.3× better overall time reduction against the dense mode when processing MobileNet.

Fig. 10 (b) introduces the energy efficiency of the sparse mode under various benchmarks. By considering data reuse and space reuse, energy consumption has also been reduced. The sparse mode achieves 1.2×, 1.2× and 1.4× less energy consumption than the dense mode in MobileNet pointwise

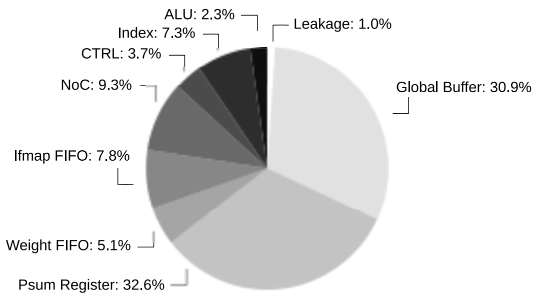

**FIGURE 11.** On-chip energy breakdown in sparse Alexnet.

conv layer 2, pointwise conv layer 13, and pointwise conv layer 14, respectively. The sparse mode's energy efficiency is 5173.9 inferences/J for sparse MobileNet, which achieves 1.2× better overall energy reduction against the dense mode as 4206.4 inferences/J when processing dense MobileNet. The dense mode of MOSDA suffers 1.1× larger energy overhead than the architecture in Reference [32] in MobileNet.

Three state-of-the-art DNN accelerators [13], [18], [25] are compared with MOSDA in Tab. 9. EIE [25] and OuterSPACE [18] are specialized accelerators designed for FC layers and matrix multiplications. Eyeriss v2 [13] is an energy-efficient sparse CNN accelerator which is able to process both FC layers and CONV layers. In order to exploit more data reuse with PEs, Eyeriss v2 has the $[s-s]$ matching type in its scratchpad memories in each PE. The evaluation results on several sparse neural networks are introduced in Tab. 9. Based on the gate-level simulation of sparse Alexnet, the average throughput of MOSDA based $[a-a]$ dataflow when processing the sparse Alexnet is 315.3 inferences/second on average, which achieves 1.1× better improvement with 256 multipliers than the $[s-s]$ dataflow with 384 multipliers. That is because the matching type $[a-a]$ allows MOSDA to skip zero data more efficiently, thereby enhancing throughput. Similarly, although MOSDA suffers from a large accumulation overhead, it still achieves 1507.8 inferences/J, which has 2.1× better energy efficiency than the $[s-s]$ processing dataflow in sparse Alexnet. The reason is divided into two parts. First, MOSDA does not store many weights within registers in each PE, which reduces the access energy to each register. Second, $[a-a]$ processing dataflow guarantees all loaded data in registers generate meaningful results, thereby reducing the number of data movement. We further utilize sparse Resnet-50 [41] and sparse MobileNet v2 [42] with width multiplier of 1.0 and input size of 224 × 224 with data from ImageNet dataset to prove the effectiveness of MOSDA. According to post-layout simulations, MOSDA achieves 493.1 inferences/second and 839.9 inferences/second when processing Resnet-50 and MobileNet v2, respectively. MOSDA achieves 2764.4 inferences/J and 5574.6 inferences/J when processing Resnet-50 and MobileNet v2, respectively.

This article mainly focuses on the sparsity within one frame, which is also called spatial sparsity. Besides the spatial sparsity, there is also temporal sparsity among frames [43]. If data rarely change over time, data is regarded as temporarily sparse. A large number of data movement is able to be reduced by utilizing temporal sparsity. MOSDA is also available for the temporal sparsity since the matching type over frames is also [a-a].

## VI. CONCLUSION

This article proposes the processing property as the matching type for the PE utilization issue in sparse DNN processing. Focusing on three processing properties as the date reuse, the space reuse and the matching type, this article proposes an efficient general-purpose hardware accelerator called MOSDA, which speeds up the processing by skipping zero data without complex control logic in the sparse DNN processing. According to our case study, MOSDA outperforms the state-of-the-art CNN accelerator with 1.1× time reduction and 2.1× energy reduction in sparse Alexnet.

## REFERENCES

[1] L. Yann, B. Yoshua, and H. Geoffrey, "Deep learning," *Nature*, vol. 521, no. 1, pp. 436–444, May 2015.
[2] K. Vissers, "The Xilinx adaptive compute acceleration platform (ACAP)," in *Proc. ACM/SIGDA Int. Symp. Field Programmable Gate Arrays*, New York, NY, USA, 2019, p. 83.
[3] Intel. (2019). *Neural Compute Stick 2*. [Online]. Available: https://software.intel.com/en-us/articles/OpenVINO-RelNotes
[4] Google. (2019). *Edge TPU*. [Online]. Available: https://cloud.google.com/edge-tpu/
[5] D. Silver *et al.*, "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, no. 484, pp. 484–489, Jan. 2016.
[6] A. G. Howard *et al.*, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," 2017. [Online]. Available: arXiv:1704.04861.
[7] C. Szegedy *et al.*, "Going deeper with convolutions," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2015, pp. 1–9.
[8] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proc. IEEE*, vol. 105, no. 12, pp. 2295–2329, Dec. 2017.
[9] B. L. Deng, G. Li, S. Han, L. Shi, and Y. Xie, "Model compression and hardware acceleration for neural networks: A comprehensive survey," *Proc. IEEE*, vol. 108, no. 4, pp. 485–532, Apr. 2020.
[10] 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," *Proc. IEEE Int. Solid-State Circuits Conf. (ISSCC)*, 2018, pp. 218–220.
[11] N. P. Jouppi *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proc. ACM/IEEE 44th Annu. Int. Symp. Comput. Arch. (ISCA)*, Jun. 2017, pp. 1–12.
[12] B. Moons and M. Verhelst, "An energy-efficient precision-scalable convnet processor in 40-nm CMOS," *IEEE J. Solid-State Circuits*, vol. 52, no. 4, pp. 903–914, Apr. 2017.
[13] Y.-H. Chen, T.-J. Yang, J. Emer, and V. Sze, "Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices," *IEEE Trans. Emerg. Sel. Topics Circuits Syst.*, vol. 9, no. 2, pp. 292–308, Jun. 2019.
[14] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: Ineffectual-neuron-free deep neural network computing," in *Proc. ACM/IEEE 43rd Annu. Int. Symp. Comput. Arch. (ISCA)*, Jun. 2016, pp. 1–13.
[15] Z. Du *et al.*, "ShiDianNao: Shifting vision processing closer to the sensor," in *Proc. ACM/IEEE 42nd Annu. Int. Symp. Comput. Arch. (ISCA)*, Jun. 2015, pp. 92–104.
[16] S. Yin *et al.*, "A 1.06-to-5.09 TOPS/W reconfigurable hybrid-neural-network processor for deep learning applications," in *Proc. Symp. VLSI Circuits*, 2017, pp. C26–C27.
[17] M. Cho and D. Brand, "MEC: Memory-efficient convolution for deep neural network," 2017. [Online]. Available: arXiv:1706.06873.

[18] S. Pal *et al.*, "A 7.3 M output non-zeros/J sparse matrix-matrix multiplication accelerator using memory reconfiguration in 40 nm," in *Proc. Symp. VLSI Technol.*, Jun. 2019, pp. C150–C151.
[19] S. Han, J. Pool, J. Tran, and W. J. Dally, "Learning both weights and connections for efficient neural network," in *Proc. Adv. Neural Inf. Process. Syst.*. 2015, pp. 1135–1143.
[20] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size," 2016. [Online]. Available: arXiv:1602.07360.
[21] S. Anwar, K. Hwang, and W. Sung, "Structured pruning of deep convolutional neural networks," *J. Emerg. Technol. Comput. Syst.*, vol. 13, no. 3, p. 32, Feb. 2017.
[22] T.-J. Yang, Y.-H. Chen, and V. Sze, "Designing energy-efficient convolutional neural networks using energy-aware pruning," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, 2017, pp. 6071–6079.
[23] T. Zhang *et al.*, "A systematic DNN weight pruning framework using alternating direction method of multipliers," in *Computer Vision—ECCV 2018*. Cham, Switzerland: Springer Int. Publ., 2018, pp. 191–207.
[24] G. Venkatesh, E. Nurvitadhi, and D. Marr, "Accelerating deep convolutional networks using low-precision and sparsity," in *Proc. IEEE Int. Conf. Acoust. Speech Signal Process. (ICASSP)*, 2017, pp. 2861–2865.
[25] S. Han *et al.*, "EIE: Efficient inference engine on compressed deep neural network," in *Proc. ACM/IEEE 43rd Annu. Int. Symp. Comput. Arch. (ISCA)*, Jun. 2016, pp. 243–254.
[26] O. Moreira *et al.*, "Neuronflow: A hybrid neuromorphic—Dataflow processor architecture for AI workloads," in *Proc. 2nd IEEE Int. Conf. Artif. Intell. Circuits Syst. (AICAS)*, 2020, pp. 1–5.
[27] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," Sep. 2014. [Online]. Available: arXiv 1409.1556.
[28] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE J. Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, Jan. 2017.
[29] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proc. IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov. 1998.
[30] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2016, pp. 770–778.
[31] A. Parashar *et al.*, "Timeloop: A systematic approach to DNN accelerator evaluation," in *Proc. IEEE Int. Symp. Performance Anal. Syst. Softw. (ISPASS)*. Madison, WI, USA, Mar. 2019, pp. 304–315. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ISPASS.2019.00042
[32] H. Xu, J. Shiomi, and H. Onodera, "On-chip memory optimized CNN accelerator with efficient partial-sum accumulation," in *Proc. Great Lakes Symp. VLSI*, Beijing, China, 2020, pp. 21–26.
[33] Y. Saad, *SPARSKIT: A Basic Tool Kit for Sparse Matrix Computations—Version 2*, Natl. Aeronautics Space Admin. NASA, Washington, DC, USA, 1994.
[34] A. Parashar *et al.*, "SCNN: An accelerator for compressed-sparse convolutional neural networks," *ACM SIGARCH Comput. Archit. News*, vol. 45, no. 2, pp. 27–40, Jun. 2017. [Online]. Available: https://doi.org/10.1145/3140659.3080254
[35] Z. Yuan *et al.*, "STICKER: An energy-efficient multi-sparsity compatible accelerator for convolutional neural networks in 65-nm CMOS," *IEEE J. Solid-State Circuits*, vol. 55, no. 2, pp. 465–477, Feb. 2020.
[36] S. J. E. Wilton and N. P. Jouppi, "CACTI: An enhanced cache access and cycle time model," *IEEE J. Solid-State Circuits*, vol. 31, no. 5, pp. 677–688, May 1996.
[37] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds. Red Hook, NY, USA: Curran Associates, Inc., 2012, pp. 1097–1105.
[38] O. Russakovsky *et al.*, "ImageNet large scale visual recognition challenge," *Int. J. Comput. Vis. (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015.
[39] J. Leskovec and R. Sosič, "SNAP: A general-purpose network analysis and graph-mining library," *ACM Trans. Intell. Syst. Technol.*, vol. 8, no. 1, p. 1, 2016.
[40] M. Zhu and S. Gupta, "To prune, or not to prune: Exploring the efficacy of pruning for model compression," 2018. [Online]. Available: arXiv:1710.01878.
[41] T. Gale, E. Elsen, and S. Hooker, "The state of sparsity in deep neural networks," 2019. [Online]. Available: http://arxiv.org/abs/1902.09574.
[42] E. Elsen, M. Dukhan, T. Gale, and K. Simonyan, "Fast sparse convnets," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR)*. Los Alamitos, CA, USA, Jun. 2020, pp. 14617–14626. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/CVPR42600.2020.01464
[43] A. Yousefzadeh *et al.*, "Asynchronous spiking neurons, the natural key to exploit temporal sparsity," *IEEE J. Emerg. Sel. Topics Circuits Syst.*, vol. 9, no. 4, pp. 668–678, Dec. 2019.

**HONGJIE XU** (Student Member, IEEE) received the B.E. degree in microelectronics from the University of Electronic Science and Technology of China, Chengdu, China, in 2016, and the M.E. degree in informatics from Kyoto University, Kyoto, Japan, in 2019, where he is currently pursing the Ph.D. degree with the Department of Communications and Computer Engineering, Graduate School of Informatics. Since 2019, he has been a Research Assistant with Doctoral Program for World-Leading Innovative and Smart Education, Ministry of Education, Culture, Sports, Science and Technology. His research interest includes energy-efficient VLSI system design.

**JUN SHIOMI** (Member, IEEE) received the B.E. degree in electrical and electronics engineering, the M.E. degree in communications and computer engineering, and the Ph.D. degree in informatics from Kyoto University, Kyoto, Japan, in 2014, 2016, and 2017, respectively. From 2016 to 2017, he was a Research Fellow with the Japan Society for the Promotion of Science. Since 2017, he has been an Assistant Professor with the Department of Communications and Computer Engineering, Graduate School of Informatics, Kyoto University. His research interests include modeling and computer-aided design for low-power and low-voltage system-on-chips.

**HIDETOSHI ONODERA** (Fellow, IEEE) received the B.E., M.E., and Dr.Eng. degrees in electronic engineering from Kyoto University, Kyoto, Japan, in 1978, 1980, and 1984, respectively. In 1983, he joined the Department of Electronics, Kyoto University, where he is currently a Professor with the Department of Communications and Computer Engineering, Graduate School of Informatics. His research interests include design technologies for digital, analog, RF LSIs, with particular emphasis on low-power design, design for manufacturability, and design for dependability. He served as the Program Chair and the General Chair of ICCAD and ASPDAC. He was the Chairman of the IPSJ SIG-SLDM (System LSI Design Methodology), the IEICE Technical Group on VLSI Design Technologies, the IEEE SSCS Kansai Chapter, and the IEEE CASS Kansai Chapter. He has served as the Editor-in-Chief for *IEICE Transactions on Electronics* and *IPSJ Transactions on System LSI Design Methodology*. He is an IEICE Fellow and a member of the Science Council of Japan.