PAPER Special Section on Low-Power and High-Speed Chips

In Search of the Performance- and Energy-Efficient CNN Accelerators*

Stanislav SEDUKHIN[†], ***Nonmember***, **Yoichi TOMIOKA**[†a)], ***and*** **Kohei YAMAMOTO**[††], ***Members***

SUMMARY In this paper, starting from the algorithm, a performance- and energy-efficient 3D structure or shape of the Tensor Processing Engine (TPE) for CNN acceleration is systematically searched and evaluated. An optimal accelerator's shape maximizes the number of concurrent MAC operations per clock cycle while minimizes the number of redundant operations. The proposed 3D vector-parallel TPE architecture with an optimal shape can be very efficiently used for considerable CNN acceleration. Due to implemented support of inter-block image data independency, it is possible to use multiple of such TPEs for the additional CNN acceleration. Moreover, it is shown that the proposed TPE can also be uniformly used for acceleration of the different CNN models such as VGG, ResNet, YOLO, and SSD. We also demonstrate that our theoretical efficiency analysis is matched with the result of a real implementation for an SSD model to which a state-of-the-art channel pruning technique is applied.

key words: *multi-channel convolution, tensor processing, vector-parallel computing, data reusing, computing efficiency*

1. Introduction

Deep neural networks (DNNs) have recently demonstrated their extraordinary ability to make predictions on large amounts of data with very high accuracy. A very popular class of DNNs, commonly used to analyze visual imagery, is multi-layer convolutional neural networks (CNNs) where basic processing represents the layer-to-layer changeable multi-channel 2D convolution. For some CNNs, this type of convolution represents more than 90% of the total workload [1]–[3]. Each such convolution requires a massive execution of the same scalar multiply-accumulate (MAC) operation on the multi-dimensional tensor data [4]. A native algorithm for multi-channel 2D convolution exhibits an enormous concurrency of MAC operations, and therefore, hardware accelerators with a big number of MAC units are a natural solution to such computational requirements imposed by CNNs.

As soon as it was recognized that a multi-channel 2D convolution used in CNN can be effectively converted to the general matrix-by-matrix multiply-add (GEMM) [5]–[7], the previously well-known systolic array processing (SAP) [8], [9] began to be widely used in designing CNN accelerators. For example, Google, in a few generations of Tensor Processing Units (TPUs) [10]–[13], has used the sizeable systolic array architectures to greatly accelerate convolution operations. These accelerators can be efficiently used in various stages of the machine learning: whether in training or inference, on edge devices or on the cloud [14].

The main advantages of SAP, which are favorable to VLSI systems [15], [16], can be listed as follows: (1) homogeneity and simplicity of processing elements (PEs) to implement MAC operation; (2) a structural regularity and planarity of processing; (3) a locality of mesh-like PEs interconnect; (4) a deep data reuse by data flows across the array processor which drastically reduces the number of expensive memory references.

However, despite these beneficial factors, SAP has substantial disadvantages which may limit a practical usage for some applications. Some of these disadvantages are: (1) relatively big latency due to pipelining of processing via data streaming through the array of simple PEs (the latency is proportional to the size of systolic array); (2) even if all 2D data are available and ready for processing, e.g., all pixels in an image, such matrix data should be pre-arranged and loaded vector-by-vector through peripheral PEs into a planar systolic array. To reduce a relatively big startup latency of systolic array processing, but keep the same throughput of computing, i.e., keep the same number of concurrent MAC operations per clock cycle, Google has changed the size and number of the systolic-based Matrix Multiply Units (MXUs) in the different generations of TPUs from a single huge 256×256 array of 8-bit MAC units [10] in the TPUv1 to the four smaller 128×128 arrays of MXUs per chip in the two latest TPU versions [13], [46]. Note that pipelining of vector-by-vector distribution in a planar systolic array processor has been replaced with a vector-by-vector broadcast [17], [18] in the non-systolic Tesla's Full-Self-Driving (FSD) chip which totally eliminate the startup latency and mesh-like interconnect of systolic processing [3], [19].

In this paper we are going to eliminate or reduce the SAP disadvantages while keeping the main advantages by, firstly, assuming that all data are available for computing and, secondly, data are reused inside array processor not by streaming across PEs but by circulation amongst PEs such that all resulting data remains inside an array processor and ready for the further computing and reusing. Moreover, we will demonstrate the preferences of CNN acceleration by

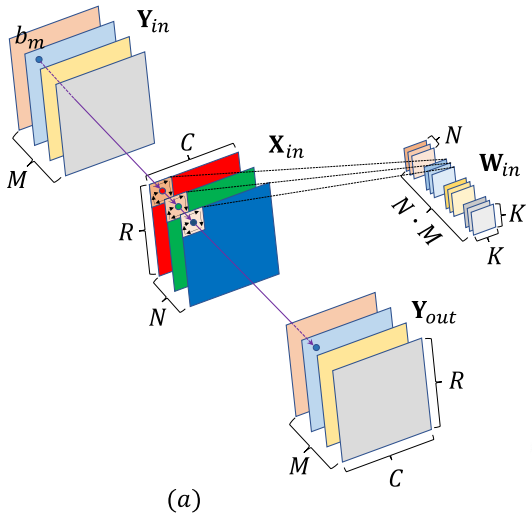
Manuscript received June 30, 2021.
Manuscript revised October 19, 2021.
Manuscript publicized December 3, 2021.

[†]The authors are with the University of Aizu, Aizu-Wakamatsu-shi, 965–8580 Japan.
[††]The author is with the Oki Electric Industry Co., Ltd., Osaka-shi, 541–0051 Japan.

*This paper was presented at the 2021 IEEE Symposium in Low-Power and High-Speed Chips (COOL CHIPS).

a) E-mail: ytomioka@u-aizu.ac.jp (Corresponding author)

DOI: 10.1587/transele.2021LHP0003




```
/* Consecutively for N input channels
for n ∈ {1, 2, ..., N} do
    /* Concurrently for all the M output channels
    forall m ∈ {1, 2, ..., M} do
        /* Concurrently for all the R × C output pixels
        forall (r, c) ∈ (1 : R, 1 : C) do
            /* Initialization of y[m, r, c]
            y[m, r, c] ← if bias is given then b[m] else if residual then y[m, r, c] else 0
            /* An element-wise loop through the 2D K × K kernel
            foreach (i, j) ∈ (1 : K, 1 : K) do
                /* A single scalar MAC operation
                y[m, r, c] ← x[n, r + i, c + j] · w[m, n, i, j] + y[m, r, c]
            end
        end
    end
end
```

(b)

Fig. 1 The data used in the multi-channel 2D convolution (a) and an execution program (b) which keeps an inherent in Eq. (1) 3-D ($M \times R \times C$) MAC-concurrency for the given convolutional layer.

using not GEMM-based planar approach, but by direct 3D implementation of a multi-channel 2D convolution which does not require data rearrangement. Another our target is to find an optimal single structure or 3D shape of CNN accelerator which maximizes the number of concurrent MAC results per clock cycle while minimizing the number of redundant MAC operations when implementing *all* layer-by-layer multi-channel convolutions for any given CNN model.

2. Multi-Channel 2D Convolution

2.1 General Equation and Operational Concurrency

A multi-channel 2D convolution can be defined for each output element by the following output-centric [20] equation:

$$\forall (m, r, c) \in \{1 : M\} \times \{1 : R\} \times \{1 : C\} :$$

$$\mathbf{Y}_{m,r,c} = \sum_{n\in}^{\{1:N\}} \sum_{i\in}^{\{1:K\}} \sum_{j\in}^{\{1:K\}} \mathbf{X}_{n,r+i,c+j} \cdot \mathbf{W}_{n,m,i,j} + \mathbf{b}_m, \qquad (1)$$

where $\mathbf{Y}_{M\times R\times C}$ is a 3-dimensional (3D) tensor of output feature maps (OFMs), $\mathbf{X}_{N\times R\times C}$ is a 3D tensor of input feature maps (IFMs), $\mathbf{W}_{N\times M\times K\times K}$ is a 4D tensor of given parameters or learned filter coefficients (weights), $\mathbf{b}_M$ is a 1D vector of biases, N and M are the number of input and output channels, respectively, and the filter window size is $K \times K$. Note that the order of summations in Eq. (1) is not explicitly defined, i.e., any order is permissible.

The data used in the multi-channel 2D convolution are shown schematically in Fig. 1(a) where the spatial sizes of input and output feature maps are usually making equal by proper zero padding of IFM. The size of padding is computed as $p = (K - 1)/2$. Note that for each CONV layer the input OFMs $\mathbf{Y}_{in}$ should initially be equal to either a *zero tensor* or populated by a *bias tensor* for the first layer or an *OFM tensor* from the previous layer to implement a possible residual operation or short-cut connection [21], [22] (see Fig. 1(a)).

The ∀-quantifier "for each" in Eq. (1) clearly shows independent element-wise computing of the OFM $\mathbf{Y}_{out}$ where each output $\mathbf{Y}(m, r, c)$-element requires exactly K^2N scalar multiply-accumulate (MAC) operations in the form of $y \leftarrow x \cdot w + y$. Obviously, the total number of these MAC operations to compute Eq. (1) is K^2NMRC and if each MAC operation can be executed in a single time-step or clock cycle then Eq. (1) can be computed sequentially element-by-element in K^2NMRC time-steps.

Each MAC operation requires three-read/one-write memory accesses to deliver three operands to the MAC unit and return resulting operand to the memory. This required data movement from/to four-ported memory is considered to be the most expensive part of MAC operation in terms of the latency and energy consumption [13], [23], [24]. A well-known technique to diminish this problem is a deep data reusing in each level of memory hierarchy (the main memory, caches, register files) and between MAC units (like in systolic array processors).

Note that the total number of involved data elements, i.e., the IFM, filter coefficients and initial OFM is $RCN + K^2NM + RCM$, i.e., the MAC intensity or the average number of concurrent MACs per a single three-operand data access can be estimated as a ratio $\lceil (K^2NMRC)/(RCN + K^2NM + RCM) \rceil$. The MAC intensity can also be interpreted as degree of single operand reusing. For example, the MAC intensity $K^2NMRC/RCN = K^2M$ shows that each IFM-element $x(n, r, c)$ after reading from the memory can be reused in K^2M different MAC operations while the MAC intensity $K^2NMRC/K^2NM = RC$ shows the number of MAC operations which can reuse the same filter coefficient $w(n, m, i, j) \in \mathbf{W}_{in}$.

The $\forall$-quantifier also demonstrates that potentially possible MAC concurrency is MRC. Therefore, a computing of Eq. (1) can also be implemented in K^2N time-steps with the MRC intermediate MAC results per time-step. This potentially possible 3D MAC concurrency consists of the spatial $(R \times C)$ parallelism of pixels in the IFM and vector parallelism of M output channels. Computing of Eq. (1) can be expressed in the form of holistic program (see Fig. 1(b)) which keeps an $M \times R \times C$ scalar MAC concurrency inherent in Eq. (1) on each time-sep and, therefore, shows how to compute (1) in K^2N time-steps.

3. Scalable Vector-Parallel Tensor Computing

The multi-channel 2D convolution (1) can be viewed as a set of 2D convolutions for Multiple-Input and Multiple-Output (MIMO) channels where each set consists of a nested subset of 2D convolutions for Single-Input Multiple-Output (SIMO) channels which, in turn, includes the lowest level of a nested sub-sub-set of 2D convolutions for Single-Input Single-Output (SISO) channels.

3.1 SISO Channels 2D Convolution

The SISO convolution is a standard 2D convolution of a given image. For our case, it is specified as a 2D convolution of the IFM for n-th input channel, i.e., an $(R \times C)$-matrix X_n, with a 2D $(K \times K)$-filter $W_{n,m}$, resulting in computing of an $(R \times C)$-matrix Y_m for any $n \in \{1 : N\}$ and $m \in \{1 : M\}$. It is again assumed that an input matrix X_n is properly padded.

The 2D convolution can be viewed as the sum of the point-wise (1×1) convolutions computed for each input pixel in any order. The summation order can be common for all the pixels in an image by massively-parallel moving or sliding of the pixels across each other in a some predefined sequence which ensures an involvement in the summation of all needed pixels from the window-based area.

In terms of operational concurrency, this observation allows to implement an independent updating of *all* output pixels $y() \in Y_m$ by sequentially supplying the weighted neighbor pixels $x() \in X_n$ from the $(K \times K)$-window to the central to this window pixel $x(\mathrm{O})$ which can be any input pixel $x() \in X_n$ as shown in Fig. 2(a). It can be effectively implemented by cyclically [†] shifting or systolically rolling the image along a Hamiltonian path which defines a neighbor pixel-by-pixel delivery in some linear order [25], [26]. Note that it can be done either from the central pixel $x(\mathrm{O})$ all the way to the farthest in the window pixel $x(\mathrm{SE})$ or, in the opposite direction, from the pixel $x(\mathrm{SE})$ to $x(\mathrm{O})$.

Figure 2(a) demonstrates an example of the SISO 2D convolution for the filter size $K = 3$. The left side of Fig. 2(a) shows in the red color a possible Hamiltonian path of shifting or rolling an image around any pixel $x(\mathrm{O}) \in X_n$ covering all eight neighbor pixels, i.e., step-by-step shifting an image X_n in the directions of

[†]The IFM edge pixels should not be lost for further reusing.

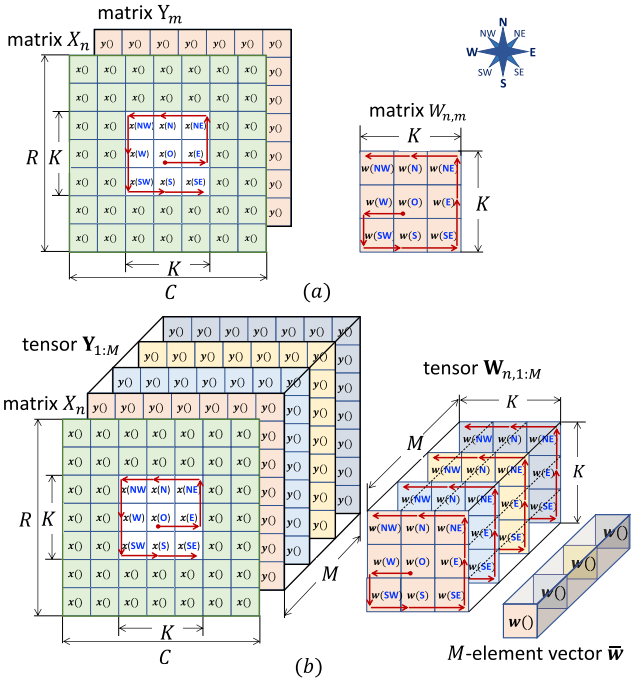


Fig. 2 The data needed to implement a 2D convolution for the SISO (a) and SIMO (b) channels.

O→E→N→W→W→S→S→E→E will provide for any location of pixel $x(\mathrm{O})$ delivery of all its neighbor pixels in the order of $x(\mathrm{W})$, $x(\mathrm{SW})$, $x(\mathrm{S})$, $x(\mathrm{SE})$, $x(\mathrm{E})$, $x(\mathrm{NE})$, $x(\mathrm{N})$, $x(\mathrm{NW})$. To correctly update every output pixel $y() \leftarrow x() \cdot w() + y()$ the corresponding filter coefficients $w() \in W_{n,m}$ should be broadcast to all shifted pixels $x()$ in the opposite order as it is shown in red on the right side of Fig. 2(a). This SISO 2D convolution can be computed in K^2 time-steps independently from the image size by using an $(R \times C)$-array of scalar MAC units with a torus interconnect for temporal reusing operands $x()$. Each time-step consists of the same processing pattern:

- **Broadcast** {scalar $w()$};
- **Update** {$(R \times C)$-matrix $Y_m = [y()]$};
- **Roll** {$(R \times C)$-matrix $X_n = [x()]$}.

3.2 SIMO Channels 2D Convolution

It is clear from the Eq. (1) that an IFM for the same input channel $n \in \{1 : N\}$ is reused for all M output channels, i.e., the same $(R \times C)$-matrix X_n is used for updating an output $(R \times C \times M)$-tensor $\mathbf{Y}_{1:M}$. This updating requires to use also a corresponding $(K \times K \times M)$-tensor of filter coefficients $\mathbf{W}_{n,1:M}$ as shown in Fig. 2(b). This vector-parallel style of computing (1) is implemented on the $(R \times C)$-array of M-element vector MAC (VMAC) units with a torus interconnect. Such organization allows to compute a SIMO 2D convolution in also K^2 time-steps while producing not planar RC, but volume of RCM intermediate MAC results per time-step [27]. Each of the RC VMAC units executes on each step a scalar-by-vector multiply-add operation $\vec{y}() \leftarrow x() \odot \vec{w}() \oplus \vec{y}()$. Each

VMAC unit has, additionally to the scalar, M-element vector operand registers to store locally the broadcast vector $\vec{w}$ and updated vector $\vec{y}$. The same processing pattern with a Hamiltonian's path to select both the vector $\vec{w}$ and direction of shifting a matrix X_n is used on each of K^2 time-steps, i.e., processing pattern for the SIMO 2D convolution is

- **Broadcast** {M-vector $\vec{w}$};
- **Update** {$(R \times C \times M)$-tensor $\mathbf{Y}_{1:M}$};
- **Roll** {$(R \times C)$-matrix X_n}.

3.3 MIMO Channels 2D Convolution

The SIMO 2D convolution can be naturally used to compute the MIMO 2D convolution by iteratively updating an OFM tensor $\mathbf{Y}_{1:M}$ for all the input channels $n \in \{1 : N\}$ [27]. For every new input channel it can be done by keeping a previously computed output tensor $\mathbf{Y}_{1:M}$ on the $(R \times C)$-array of vector registers $\vec{y}()$ while changing an IFM matrix X_n and use a corresponding tensor $\mathbf{W}_{n,1:M}$ to implement the next SIMO convolution. Note that an order of the input channel selection is not required to be fixed.

3.4 Block Convolution

The existing limitations of memory and computing resources forced to divide the $(R \times R)$ IFM into an array of $(r \times r)$ *tiles*, process each tile independently, and combine partial solutions into the final $(R \times R)$ OFM [†]. However, implementation of a kernel-wise convolution leads to an issue in computing near the edges of the tiles due to inter-tile dependencies. To address this, each individual tile should be slightly overlapped or padded by providing supplementary data at the boundary. The tile padding size is also setting as $p = (K - 1)/2$. We call this padded tile of pixel data as a *block*. A 3D $[(r+2p)\times(r+2p)\times N]$ tensor of the IFM blocks for all N input channels, which is called a *pillar*, provides all needed data to compute a single $(r \times r)$-tile of the OFM.

In terms of a tile padding, there are at least three possible methods: zero or constant padding, repeated or mirrored padding and overlapped padding. Zero padding pads the boundary pixels with zeros while repeated padding duplicating the boundary pixels outwards based on the coherency of the pixels in an image. For these two methods there is no need neither explicitly pad each tile nor rearrange memory pattern on hardware, because block padding can be merged into process of convolution either by initialization or by manipulating memory addressing, which is a memory-efficient *approximation* of original convolution [28].

The selected here overlapped padding is based on replication of the boundary pixels from the neighbor tiles such that it solves the inter-tile data dependency problem. The result of this block overlapped 2D convolution is exactly the same as non-blocked convolution of the original image, but requires additionally data rearrangement in the memory [29], [30].

[†]For simplicity, we assume $R = C$ and r is the size of a tile.

4. Accelerator's Architecture and Area-Time Analysis

4.1 A Generic Architecture of the TPE

An architecture of the accelerator can be directly obtained from the above analysis of the block multi-channel 2D convolution. This analysis covers the algorithm's complexity, operational concurrency, different ways of data reusing, blocking, etc. A generic architecture of the CONV accelerator which we call the Tensor Processing Engine (TPE) is shown in Fig. 3. The TPE is connected to the host computer which implements other than CONV operations with low data reusing like pooling, quantization, activation, data management, etc., in consideration of Amdahl's Law to properly support fast compute acceleration of the TPE.

The host connects to the TPE through a common off-chip memory and, besides, the TPE has its own on-chip memory to store blocks of the input and output data which are needed to concurrently compute an $(m \times r \times r)$ intermediate elements of the $(M \times R \times R)$ OFM tensor data, where the vector length $m \in \{1 : M\}$ and the array size $r \in \{K : R\}$.

The on-chip distributed memory with a parallel read/write data access is directly connected to the $(r \times r)$-array of Vector MAC (VMAC) units. Each VMAC unit has the scalar and m-way vector register file (RF) to implement scalar-by-vector MAC operation $\vec{y}() \leftarrow x() \cdot \vec{w}() + \vec{y}()$. It is clear that this architecture is based on the three levels of memory hierarchy: off-chip memory, on-chip memory and array of the RFs.

The scalar registers of the MAC array, storing initially a block of $x()$-operands from the IFM X_{in}, are interconnected by torus network (see Fig. 3 where wrap-around connections are not shown for clarity). The m-element vector of weights $\vec{w}()$ is replicated by broadcast from the on-chip memory to all the m-vector $\vec{w}()$-operand registers of the MAC array. This global one-to-all network is shown in Fig. 3 in the red color.

A vector-parallel computing of $(m \times r \times r)$ intermediate elements of the $(M \times R \times R)$ output tensor data directly follows an approach described in Sect. 3 which uses a "broadcast-compute-roll" processing pattern on each time-step. A proposed m-vector $\vec{y}()$-stationary approach is based on an accumulation of m-partial sums (PSUM) inside each VMAC unit (shown in Fig. 3 in the green color).

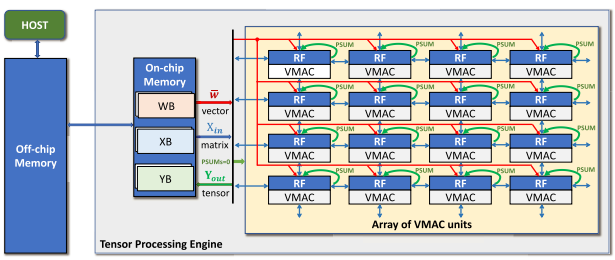


Fig. 3 A generic architecture of the Tensor Processing Engine.

4.2 Computing of the OFM "Brick" and "Pillar"

It is clear that SIMO-computing of an $(m\times r\times r)$ output data for a single input channel $n\in\{1:N\}$ and m output channels requires a planar $[(r+K-1)\times(r+K-1)]$-block of the IFM, an $(m\times K\times K)$-tensor of weights and, in general, an initial $(m\times r\times r)$-tensor of the OFMs. As it was mentioned, an initial $(m\times r\times r)$-tensor can be either the zero tensor or tensor of biases or tensor of OFMs from the previous layer. We call an $(m\times r\times r)$ output data structure a *brick* and, in general, many bricks are needed to compute a single pillar and many pillars can be required, especially for initial CONV layers in the CNN model, to complete computing of an OFM tensor. This SIMO-computing of the m-channel output brick of data for a single input channel is finished in K^2 time-steps or clock cycles.

To compute an output brick iteratively for all N input channels in the K^2N time-steps, i.e., to compute 2D convolution for the all input, but multiple output channels, the whole $[(r+K-1)\times(r+K-1)\times N]$-pillar of the IFMs blocks and $(m\times K\times K\times N)$-tensor of weights are needed as it is shown in Fig. 4(a). The next output brick in the same pillar of OFMs can then be computed in any order.

It can be seen that the same input pillar of the IFMs and all $[N\times M\times(K\times K)]$-tensor of weights are needed for computing an $(M\times r\times r)$-pillar of the OFMs for all input and all output channels (see Fig. 4(b)). The $K^2N\lceil M/m\rceil$ time-steps are required for a single output pillar and, totally, $K^2N\lceil M/m\rceil\lceil R/r\rceil^2$ time-steps to finish computing a resulting $(M\times R\times R)$-tensor of the OFMs. It is clear also that each brick in the output pillar and each pillar in the output tensor can be computed independently, i.e., in parallel. It can be used for additional acceleration of CONV layer computing.

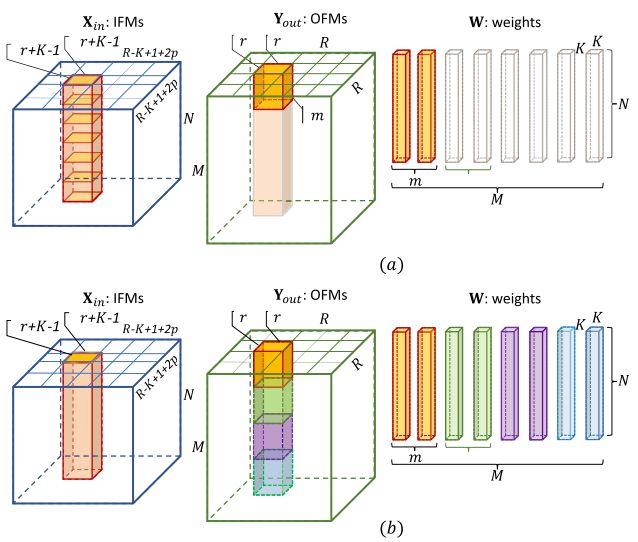


Fig. 4 Data needed for computing a brick (a) and a pillar (b) of OFMs.

4.3 Area-Time Analysis

It is possible now to make an Area-Time analysis of the proposed direct computing of multi-channel 2D convolution (1) and provide a quantitative comparison with computing of (1) by a matrix-by-matrix multiplication.

4.3.1 Direct Vector-Parallel Computing

The total number of required MAC operations to compute a single CONV layer is R^2NK^2M, which can be considered equal to the number of time-steps in sequential (serial) computing with a single MAC unit, i.e.,

$$T_1 = R^2NK^2M \tag{2}$$

time-steps are needed for computing a given CONV layer under assumption that a single three-read-one-write MAC operation is executed in one time-step or one clock cycle τ_1.

On the other hand, as it was shown above, the number of time-steps to compute a CONV layer on the TPE with the shape of $(m\times r\times r)$ is

$$T_{mr^2} = K^2N\lceil M/m\rceil\lceil R/r\rceil^2. \tag{3}$$

Here, all the mr^2 MAC results are computed in one clock cycle τ_{mr^2} with a deep data reusing. More specifically, a single m-element vector of weights $\vec{w}()$ is reused spatially by broadcast over an $(r\times r)$ array of VMAC units while all elements $x()$ of the input $[(r+K-1)\times(r+K-1)]$-block are reused temporally during K^2 time-steps through data rolling or cyclical shift across an array of VMAC units.

The speedup of a vector-parallel computing of CONV layer on the TPE over a corresponding sequential execution can be expressed as

$$S_{mr^2} = \frac{T_1}{T_{mr^2}} = \frac{M}{\lceil M/m\rceil}\cdot\left(\frac{R}{\lceil R/r\rceil}\right)^2 \le mr^2, \tag{4}$$

with the efficiency of processing

$$E_{mr^2} = \frac{S_{mr^2}}{mr^2} \le 1. \tag{5}$$

Note that, in reality, due to memory overhead, a time-step period or clock cycle for vector-parallel computing τ_{mr^2} can be perceptibly less than that of sequential computing τ_1. Moreover, for both of serial and parallel implementations, due to the different cost of data accesses in memory hierarchy [23] as well as different degrees of operation concurrency and data reusing, the clock periods of execution likely to be dissimilar for various MAC operations. These are the main reasons in discrepancy between the amount of MAC operations and real speed of processing which has been noticed in previous works [31], [32]. We, however, assume, for the simplicity of comparison that all MAC operations are similar and that $\tau_1 = \tau_{mr^2}$.

Note also that although the number of MAC operations does not directly translate to the CONV layer runtime [33], it

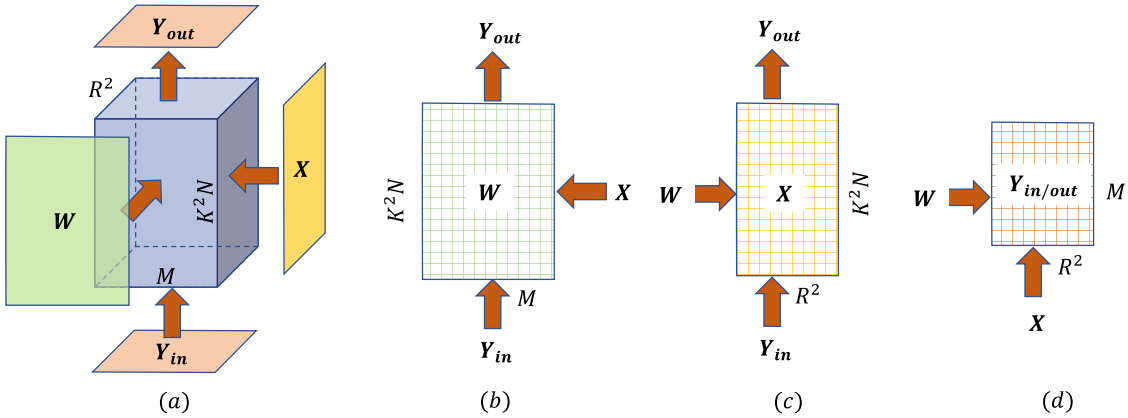


Fig. 5 The 3D index space of a matrix-by-matrix multiply-add (a) and its corresponding linear projections onto a 2D array processor space along R^2-edge (b), M-edge (c), and K^2N-edge (d).

is indicative of the runtime and allows for comparing CNN cost independent of hardware and associated software implementation details [34].

4.3.2 Convolution via Systolic Matrix Multiplication

To use a so-called GEneral Matrix Multiplication (GEMM), which is highly-optimized in software and hardware in today high-performance computers, many deep learning frameworks convert a multi-channel 2D convolution (1) to this basic operation [6], [7], [31]. In this case, reshaping of the output tensor $\mathbf{Y}_{R\times R\times M}$ into a matrix $Y_{R^2\times M}$ can be done as follows (see, e.g., [35])

$$\mathbf{Y}(m,r,c) = Y(m, r + R(c-1)).$$

Reshaping of the input tensor $\mathbf{X}_{R\times R\times N}$ into a matrix $X_{R^2\times K^2N}$ is implemented as

$$\mathbf{X}(n, r+i-1, c+j-1) = X(r+R(c-1), i+K(j-1)+K^2(n-1))$$

and, finally, reshaping the kernel tensor $\mathbf{W}_{K\times K\times N\times M}$ into a matrix $W_{K^2N\times M}$ is done by

$$\mathbf{W}(m,n,i,j) = W(m, i + K(j-1) + K^2(n-1)).$$

Using these matrices, the multi-channel convolution (1) can be computed as a matrix-by-matrix multiply-add

$$Y_{R^2\times M} = X_{R^2\times K^2N} \times W_{K^2N\times M} + Y_{R^2\times M}, \tag{6}$$

where, initially, a matrix $Y_{R^2\times M}$ is either a zero matrix or an output matrix from the previous layer.

The 3D index space of (6) as well as surrounding input/output matrices are shown in Fig. 5(a). Because each index point in a 3D $K^2N\times R^2\times M$ computational index space is associated with a single MAC operation, the total number of such operations to compute (6) is the same as for the direct computing of (1). However, a matrix multiply-add approach requires K^2 more elements in an input matrix X, i.e., more memory accesses would be needed.

The 2D array processors as the different sets of MAC units as well as the initial data flows for matrices X, W and Y can be formally obtained by linear projections of a 3D index space into a 2D processor space [16], [36]–[38]. Figures 5(b), (c), (d) demonstrate three such projections of a given 3D index space along the R^2-edge, M-edge, and K^2N-edge for the so-called weight-stationary, input-stationary, and output-stationary variants of the planar array processors, respectively.

For the systolic time-space scheduling of the MAC operations and data movement inside the 3D index space, the total number of time-steps or clock cycles to implement computing (6) is equal to the length of a critical path, i.e., $K^2N + R^2 + M - 2$, which is preserved for all the 3D→2D projections [39], [40]. However, the startup delay and computing latency, measured also in the time-steps, will be different for the different projections. Indeed, for the weight-stationary projection, a startup delay is $K^2N + M - 2$ while computing latency is R^2; for the input-stationary, a startup delay is $R^2 + K^2N - 2$ and computing delay is M; for the output-stationary, a startup delay is $R^2 + M - 2$ and computing latency is K^2N.

The all three systolic array processors have the same potential speedup of computing

$$S_{\text{systolic}} = K^2NR^2M/(K^2N + R^2 + M - 2),$$

but different efficiency or utilization levels: $R^2/(K^2N + R^2 + M - 2)$ for the weight-stationary variant, $M/(K^2N + R^2 + M - 2)$ for the input-stationary, and $K^2N/(K^2N + R^2 + M - 2)$ for the output-stationary variant. Note that our proposal also in the case of unlimited parallelism has the potential speedup

$$S_{\text{direct}} = MR^2$$

and, therefore, it has the highest possible efficiency of 100%.

For the CNN VGG16 (see Table 1 below), Fig. 6 shows the speedup (a) and efficiency (b) of the different systolic and direct implementations of (1). As it can be seen, the speedup of a direct computing is always higher than any systolic implementation, Fig. 6(a), while an efficiency is sufficiently varied not only for the differently projected systolic arrays but also for the different CNN layers, Fig. 6(b).

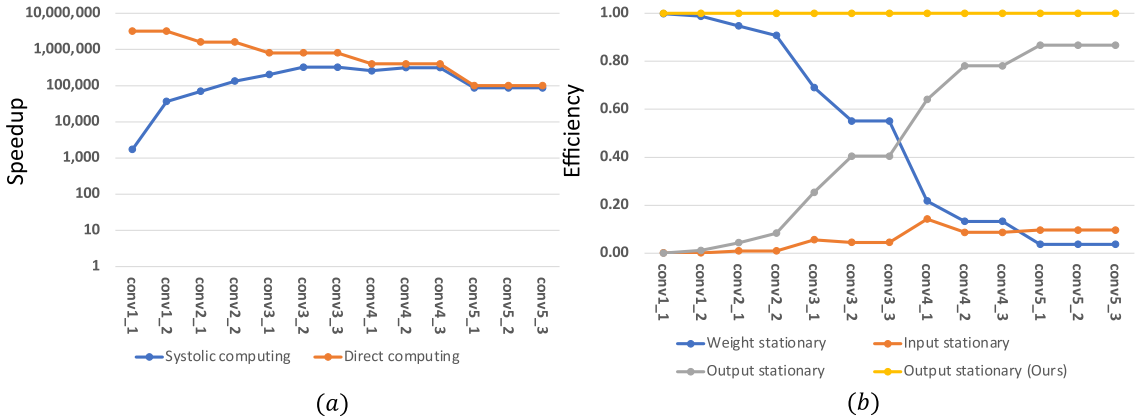


Fig. 6 Computing speedup (*a*) and efficiency of the direct and different systolic implementations (*b*).

Indeed, an efficiency of the weight-stationary systolic array processor is changing from 99.8% for the first CONV layer to 3.7% on the last three CONV layers, while for the output-stationary array processor, an efficiency is varied from 0.05% at the beginning to 86.7% at the end of processing. The input-stationary variant never reaches a computing efficiency higher than 10%.

Obviously, to keep a high efficiency of the systolic VGG16 computing, it is possible to use a weight-stationary systolic array processing from the beginning until the `conv3_3` layer and then switch to the output-stationary systolic processing. But even in this combined case, due to the inherent startup delay, a systolic array processing will never reach the highest possible efficiency of computing as in the direct implementation.

4.4 Selection of TPE's Shape for a Single CONV Layer

As it can be seen, the selection of a proper vector-parallel $(m \times r \times r)$-shape of the 3D TPE for a single CONV layer is crucial with respect to the achievable levels of performance (shown as Eq. (4)) and data reusing (shown, partially, as an efficiency). Figure 7 exhibits a solution space for selection of the TPE's shape for a single CONV layer. The three cases are shown: (a) a serial implementation on the TPE with one scalar MAC unit, i.e., when $m = r = 1$ (in the blue color); (b) an extremely vector-parallel implementation when the vector length $m = M$ and the array size $r = R$ (in the red color); (c) a tiled vector-parallel implementation for the vector length $m < M$ and the array size $K \le r < R$ (in the green color).

An area $A_{r,m}$ of each rectangle with an $(m \times r \times r)$ TPE's shape is proportional to the number of time-steps required for this specific implementation of the CONV layer. Obviously, for the extreme cases (a) and (b) we have an equality $A_{1,1} = A_{R,M}$. However, for the case (c), due to a "ceiling" function $\lceil x \rceil = \min\{n \in \mathbb{Z} | n \ge x\} = n$ in Eq. (4), an area $A_{r,m}$ may be greater than $A_{1,1} = A_{R,M}$, i.e., the TPE with this shape may compute a number of unnecessary or redundant MAC operations. The number of these operations can be estimated as $A_{1,1} - A_{r,m}$. The lower computing efficiency

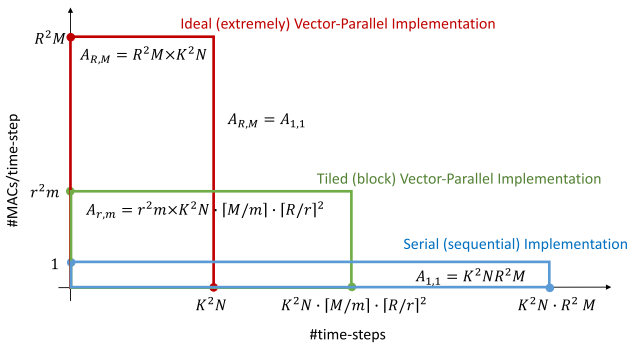


Fig. 7 An area-time solution space of the scalable TPEs.

means that more of such unnecessary MAC operations are executed.

It is important to note that if the "ceiling" functions $\lceil x \rceil$ in Eq. (4) do not have remainders, concurrently for both $x = M/m_*$ and $x = R/r_*$, the area of such tiled vector-parallel implementation A_{r_*,m_*} would be equal to $A_{1,1} = A_{M,R}$. In this case, the speedup of computing on the TPE with such an $(m_* \times r_* \times r_*)$-shape is $S_{m_* r_*^2} = m_* r_*^2$ with the maximal possible efficiency of computing $E_{m_* r_*^2} = 1$. Otherwise, the TPE executes the redundant MAC operations which will affect the speedup and efficiency.

Traditionally, the redundant MAC operations can be masked and isolated from the main data processing path, but it would require additional logic, control and energy consumption [†]. A masking of the vector MAC operations is relatively simple and straightforward, but its implementation in the fixed VMAC array might not be trivial because this array executes not only computing, but also supports data reusing by cyclical data shift across locally interconnected VMAC units.

4.5 Selection of a Single TPE's Shape for All CONV Layers

The selection of a proper TPE's shape becomes even more

[†]Including, e.g., a static energy of the masked MAC units.

Table 1 CNN VGG16

Layer	R	N	M	K	# MACs	# Params
conv1_1	224	3	64	3	86,704,128	1,728
conv1_2	224	64	64	3	1,849,688,064	36,864
conv2_1	112	64	128	3	924,844,032	73,728
conv2_2	112	128	128	3	1,849,688,064	147,456
conv3_1	56	128	256	3	924,844,032	294,912
conv3_2	56	256	256	3	1,849,688,064	589,824
conv3_3	56	256	256	3	1,849,688,064	589,824
conv4_1	28	256	512	3	924,844,032	1,179,648
conv4_2	28	512	512	3	1,849,688,064	2,359,296
conv4_3	28	512	512	3	849,688,064	2,359,296
conv5_1	14	512	512	3	462,422,016	2,359,296
conv5_2	14	512	512	3	462,422,016	2,359,296
conv5_3	14	512	512	3	462,422,016	2,359,296
Total					15,346,630,656	14,710,464

complex under a run-time variety of the CONV layer sizes in the same CNN model. In the existing state-of-the-art CNNs all the CONV layer parameters $\{R, N, M, K\}$ can be changed from layer to layer and the number of such layers can be from a few to many tens and even hundreds. In our case, a dynamic vector length (m) and array size (r) adjustment by the run-time TPE reconfiguration requires the additional processing/memory logic, time and power which will affect a computing efficiency. In existing CNNs the spatial dimension (R) is gradually shrunk while the channel dimension (M) is expanded over layers, for example, from the initial input data shape ($224 \times 224 \times 3$) to ($14 \times 14 \times 512$).

As an example, the parameters of each of the 13 convolutional layers in the VGG16 CNN as well as the total number of required MAC operations and filter coefficients are shown in Table 1.

4.5.1 Selection of the Optimal Array Size

The total number of MAC operations shows simultaneously the total number of time-steps which are needed to execute all the CONV layers on a computer with the single MAC unit. This number can be evaluated as

$$T_1^* = \sum_{l=1}^{L} T_1^{(l)},$$

where L is the total number of CONV layers ($L = 13$ in the VGG16) and $T_1^{(l)} = R_{(l)}^2 N_{(l)} K_{(l)}^2 M_{(l)}$ is a sequential or serial time complexity of the layer l.

Now we can calculate the number of time-steps

$$T_{1,r}^{(l)} = K_{(l)}^2 N_{(l)} M_{(l)} \lceil R_{(l)}/r \rceil^2$$

required for parallel, but not vector, execution of each layer $l \in \{1, 2, ..., L\}$ on the planar TPE with an $(1 \times r \times r)$-shape, i.e., $m = 1$, for the all possible array sizes $K \le r \le R_{max}$, where $R_{max} = 224$ for our case (see Table 1).

After that the total number of time-steps required for execution of all CONV layers on the same planar TPE can be computed as $T_{1,r}^* = \sum_{l=1}^{L} T_{1,r}^{(l)}$. Then the speedup $S_{1,r}^* = T_1^*/T_{1,r}^* \le r^2$ and efficiency $E_{1,r}^* = S_{1,r}^*/r^2 \le 1$ are computed also for the all possible array sizes $K \le r \le R_{max}$. Finally, for this part, the optimal array size r^* is selected for both the efficiency of processing $E_{1,r^*}^* = \max_{\forall r}\{E_{1,r}^*\}$ and time of computing $T_{1,r^*}^* = \min_{\forall r}\{T_{1,r}^*\}$. It is clear from the Table 1 that for the VGG16, $r^* = 14$ which is a greatest common divisor (GCD) for all sizes R.

Table 2 Vector length m^* and array size r^* for the different CNNs

CNN	T_1^*	m^*	r^*	T^*	S^*	E^*
VGG16 [41]	15,346,630,656	64	14	1,223,424	12,544	**1.00**
		128	14	688,896	22,277	0.89
VGG19 [1]	19,508,428,800	64	14	1,555,200	12,544	**1.00**
		128	14	854,784	22,823	0.91
ResNet50 [21]	12,649,168,896	64	14	1,008,384	12,544	**1.00**
		128	14	584,448	21,643	0.86
YOLOv3 [42]	7,415,529,472	32	8	3,620,864	2,048	**1.00**
		64	8	1,966,080	3,911	0.92
SSD300 [2]	30,129,032,192	64	19	1,351,936	22,286	0.97
		128	19	753,152	40,004	0.87
PCAS/SSD [43]	10,493,445,025	11	19	2,836,687	3,699	0.93
		32	19	1,080,760	9,709	0.84

A presence of the GCD $\neq 1$ guarantees the maximal possible efficiency of processing. But this selection is not so obvious for the other CNNs, like SSD300 [2], which have GCD $= 1$ for the input feature maps resized by pooling. Note that selection of the bigger array size with, however, less compute efficiency can be practically justified, e.g., it can be shown that $r = 28$ with the efficiency of 0.79 can also be selected for the VGG16 to further acceleration of computing.

4.5.2 Selection of the Optimal Vector Length

After selection of the optimal array size r^*, a search of the optimal vector length m^* for the array $(m \times r^* \times r^*)$ is performed for all different m. The number of time-steps required for processing a single CONV layer (l) is computed as

$$T_{m,r^*}^{(l)} = K_{(l)}^2 N_{(l)} \lceil M_{(l)}/m \rceil \lceil R_{(l)}/r^* \rceil^2.$$

Then the total number of time-steps required to process all CONV layers in the given CNN computed as $T_{m,r^*}^* = \sum_{l=1}^{L} T_{m,r^*}^{(l)}$. The corresponding speedup $S_{m,r^*}^* = T_1^*/T_{m,r^*}^* \le m(r^*)^2$ and efficiency $E_{m,r^*}^* = S_{m,r^*}^*/m(r^*)^2 \le 1$ for TPE with an $(m \times r^* \times r^*)$-shape are calculated. Then, for this final part, the optimal array size m^* is selected for both the efficiency of processing $E_{m^*,r^*}^* = \max_{\forall m}\{E_{m,r}^*\}$ and time of computing $T_{m^*,r^*}^* = \min_{\forall m}\{T_{m,r}^*\}$. For example, for the VGG16, as it can be seen from the Table 2, the value $m^* = 64$, which is a GCD for all sizes of the output channels M, drastically increases a MAC concurrency without any efficiency penalty.

Table 2 shows the parameters for a few popular CNNs including the number of time-steps needed for serial CONV layers execution T_1^*, the optimal as well as suboptimal vector length m^* and array size r^*, the number of time-steps T^* required for a vector-parallel processing on the TPE with an $(m^* \times r^* \times r^*)$-shape, and the corresponding speedup S^* and efficiency E^*. Note that to satisfy possible hardware limitations, such as the size of VMAC array $(r \times r)$ and the vector

length (m), it is feasible to scale a TPE's shape not only up, but also down while keeping a reasonable computing efficiency of vector-parallel processing.

As it can be seen from the Table 2 there is the optimal or suboptimal vector-parallel TPE shape which can be very efficiently used for considerable acceleration of computing all CNN layers. Moreover, due to an image inter-block independency, it is possible to use a cluster of such highly-efficient TPEs for the additional CNN acceleration. Furthermore, the single TPE or cluster of TPEs can also be utilized for efficient vector-parallel execution of the different CNN models which are collectively used in some important applications such as self-driving cars [3], [19].

5. Hardware Implementation

We implemented a CNN accelerator based on a single TPE on Xilinx Kintex KU5P FPGA board and evaluated the performance, power efficiency, and computational efficiency for PCAS/SSD [43]. The block diagram of our CNN accelerator is shown in Fig. 8. A TPE consists of $r \times r$ VMAC units with mesh interconnect. Each VMAC is composed of m scalar MAC units. Each edge VMAC unit has additional registers to store halo elements. Moreover, it has additional registers to avoid long wrap-around torus interconnect. In this implementation, we set r and m as 19 and 11, respectively, which are selected based on the analysis of Table 2.

The bit widths of activations and weights are 4 bits and 2 bits, respectively, while the bit width of accumulation registers is 12 bits. We implemented four MAC units using a DSP block of KU5P. These bit widths are determined by the evaluation using all test images from the PASCAL VOC 2007 dataset. We investigated the accuracy degeneration due to quantization and the overflow of each convolutional layer computation of PCAS/SSD with 2 bits for weight, 4 bits for activation, and 12 bits for the accumulator. Table 3 shows the mean average precision (mAP) before and after the application of quantization and PCAS. The overflow occurred only in the conv6 layer of PCAS/SSD, and the number of times of overflow occurred in PCAS/SSD computing was only three. The accuracy degradation after quantization and channel reduction was 1.5%.

The Data Manager has Xin, Win, Yout, First-In-First-Out (FIFO) for data transfer from/to an external memory via the Direct Memory Access (DMA) module with a width of 512 bit. An Xin FIFO stores a block of the input feature map data, and the word size of Xin FIFO is $(r+K-1)\times(r+K-1)\times 4$, which is 1764 bits for $K=3$ and $r=19$. The Xin FIFO receives a pillar of an input feature map with DMA read instruction. Until executing FIFO Flush instruction, the pillar is reused inside the FIFO. In other words, we can read the first channel of the pillar after we read the last channel. Note that data reusing of FIFO is realized by changing the address pointer of BRAM to avoid redundant reads and writes. A Win FIFO stores parameter and the word size of Win FIFO is $2m = 22$ bits. We

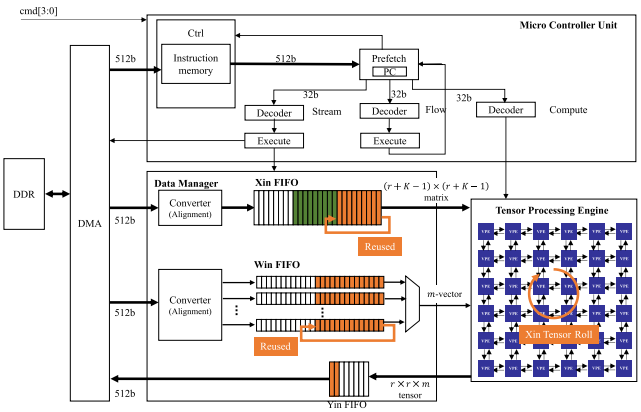


Fig. 8 Architecture of CNN Accelerator.

Table 3 Comparison between SSD300 and quantized PCAS/SSD.

Model	Weight	Activation	Accumulation	mAP
SSD300	float	float	float	72.5%
PCAS/SSD	2-bit int	4-bit uint	12-bit int	71.0%

place 24 Win FIFOs in parallel so that Win FIFOs can receive data from DMA every clock. The Win FIFOs receive all parameters of a convolutional layer with DMA read instruction, which can reuse data as in Xin FIFO. Yout FIFO is to store bricks of output feature maps calculated by TPE.

The micro controller unit (MCU) can execute DMA data read/write, loop control, and TPE execution in parallel based on the 32-bit instruction code. The TPE can reduce the waiting time for data transfer and achieve high TPE utilization by data reusing in Xin and Win FIFOs and parallel execution of data transfer and computing. Our instruction set consists of type-C, type-S, type-L, and type-H instructions. The type-C is for computing a multi-channel tensor convolution. `BRICKIS` of type-C loads a channel of a pillar of Xin FIFO, computes the convolution while broadcasting m-vector of weights, and stores the brick to the Yout FIFO iteratively. The type-S is for reading from and writing to the DDR memory. `SETST` sets the address given in 20-bit operand to address pointer specified by 3-bit operand, and `ADDST` increments the specified address pointer by the value given in 20-bit operand. `STINR` reads a pillar of the address pointer from the DDR memory and pushes it to the specified FIFO. This pillar can be reused until the specified FIFO is cleared by `FLUSH` instruction. Similarly, `STOUT` writes a specified number of bricks of the specified FIFO to the DDR memory. In type-L, `LSET` sets the value to the specified counter and `LOOP` decrements the specified counter and goes n instructions back where n is given by 16-bit operand if the specified counter is larger than zero. In type-H, `HALT` stops the program counter until receiving an awake signal. In Fig. 9, we show a code example for calculating convolution for a pillar. In this code example, NBP is the number of bricks in an output pillar, NMP is the number of 4 KB memory blocks in an input pillar, NC is the number of channels, KS is the kernel size, NMB is the number of 4 KB memory blocks in a brick. This code executes $19\cdot19\cdot11\cdot KS\cdot KS\cdot NC$

```
LSET     COUNTER0   NBP
ADDST    XinADDR    NMP
STINR    XinFIFO    XinADDR    NC
BRICKIS  19         11         NC    KS
STOUT    YoutFIFO   YoutADDR   1
ADDST    YoutADDR   NMB
LOOP     COUNTER0   3
```

Fig. 9 Example of instruction codes for computing convolution for a pillar of $(19 \times 19 \times 11) \cdot \mathrm{NBP}$.

Table 4 Implementation data

Device	KU5P
CNN Model	PCAS/SSD
LUT	120,035
FF	84,566
DSP	1,088
Max frequency [MHz]	183
Frequency [MHz]	180
Peak performance [GOPS]	1,430
Throughput [GOPS]	1,265
Power [W]	2.27
Power efficiency [GOPS/W]	629.5

MAC operations in almost $KS \cdot KS \cdot NC$ clocks. Note that `STINR` reads the next pillar from the DDR memory while `BRICKIS` computes convolution for the current pillar. We developed a compiler for the proposed hardware implementation. This compiler receives a neural network model that are compressed and quantized by PCAS and generates instruction codes. Those codes with a nested loop are loaded to the TPE's instruction memory shown in Fig. 8.

The implementation result of our CNN accelerator is shown in Table 4. The CNN Accelerator achieved a throughput of 88.5% of the peak performance, which is close to the optimal efficiency of PCAS explained in Sect. 4.5. Moreover, we achieved a power efficiency of 629.5 GOPS/W. We also show the theoretical estimation and real efficiency for each layer of PCAS/SSD in Fig. 10. The degeneration of real efficiency compared with theoretical analysis is caused by data transfer from/to external memory. However, the efficiency of our CNN accelerator is comparable with theoretical analysis in most layers. Because the data reusing rate decreases in layers where the number of input channels is relatively small, the efficiency is decreased in such layers. Moreover, the block sizes of conv9_1 to conv11_2 are 10×10 to 3×3. As a result, our TPE of $19 \times 19 \times 11$ executes many redundant MAC operations for these layers. Therefore, the efficiency of these layers is low. On the other hand, we achieve high efficiency of 88.5% across all CONV layers of PCAS/SSD.

We show the efficiency of other CNN accelerators in Table 5. The efficiencies for the all versions of TPUs are estimated by using the referenced Roofline models. As it

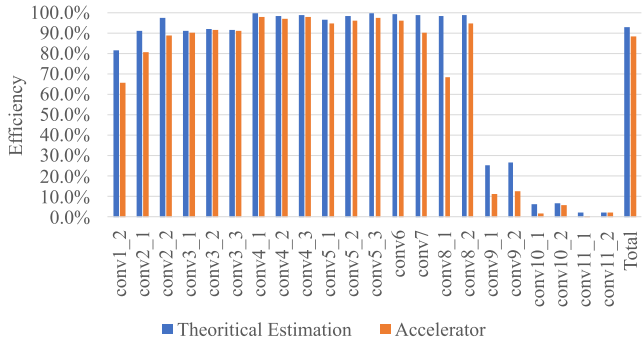


Fig. 10 Efficiency per layer for PCAS/SSD.

Table 5 MAC concurrency per clock cycle and efficiency of MAC array.

Accelerator	MACs/cc	Efficiency	CNN Model	Reference
Eyeriss	168	36%	VGG-16	[44]
ConvAix	192	76%	VGG-16	[45]
FSD	9,216	80%	Inception-v4	[19]
TPUv1	65,536	11% / 22%	CNN0 / CNN1	[12]
TPUv2	32,768	87% / 44%	CNN0 / CNN1	[46]
TPUv3	65,536	65% / 25%	CNN0 / CNN1	[46]

is mentioned in [12], CNN0 is AlphaZero, a reinforcement learning algorithm with extensive use of CNNs, which mastered the games chess, Go, and shogi while CNN1 is a Google-internal model for image recognition. A relatively low computational efficiency or utilization (33%) of the latest Google's TPU4i across the eight most utilized at Google models is also mentioned in [13]. As it can be seen our implementation and theoretical analysis of TPE demonstrate a computational efficiency that is higher or comparable with state-of-the-arts practical accelerators.

For the VGG16 model, the proposed accelerator with a TPE of 19×19×11 achieves an efficiency of 71.0% while the theoretical efficiency is 72.5%. Even though the TPE is not optimized for the VGG16 model (see Table 2), it achieves higher efficiency than Eyeriss. Also, it is comparable with ConvAix, while the number of MACs/cc is much higher. Moreover, if we select r and m optimized for VGG16, we can achieve close to the maximum efficiency and performance.

6. Conclusions and Future Work

The scalable algorithm and proper single shape 3D vector-parallel architecture of the Tensor Processing Engine for efficient layer-to-layer computing of the multi-channel 2D convolutions are proposed and evaluated. An optimal accelerator's shape maximizes the number of concurrent MAC operations per clock cycle while minimizes the number of redundant operations. The proposed 3D vector-parallel TPE architecture with an optimal shape can be very efficiently used for considerable acceleration of the CNN computing. Due to supported inter-block image data independency, it is possible to use multiple of such highly-efficient TPEs for the additional computing acceleration.

It is important to note that our approach is based on the

involvement in computing all ready for use data and, therefore, a very high memory bandwidth is required. However, we can see a few possible scenarios to soften or even eliminate this requirement.

Firstly, because of all major CNN models are computed iteratively, layer-by-layer, i.e., an output data of one layer is an input data for the next layer, it is possible to keep this data inside an array processor without sending it to the memory by implementing an output-stationary variant of computing including the CNN layers fusion [47]. Moreover, this input and output data are originally the 3D tensors and the proposed 3D array processing keeps these tensors in a natural order without destroying an integrity of data.

Secondly, an initial input of the practically big multi-channel data from the image sensors, actuators, etc., affects only the first CONV layer in a CNN model. A processing of this layer might be tightly coupled with a tensor data acquisition, e.g., by fusion of the array processor with a smart or intelligent image sensor [48] which has a highly-parallel pixels read-out [49].

And last but not least, a blocking of the input multi-dimensional data tensor (see Sect. 3.4) divides a convolution problem into the set of totally independent subproblems with the smaller size reflecting the hardware limitations such as admissible operational concurrency, memory size and bandwidth, I/O capabilities, etc. Computing of each such subproblem can be efficiently overlapped with an input/output of relatively small data tensors.

Note also that a possible extension of the proposed vector-parallel computing to other less compute-intensive types of CNN operations such as activation, pooling, quantization and others, which are currently assumed to be executed on the host computer, but can be effectively combined and implemented inside a proposed VMAC array, are considered as a future work.

Acknowledgements

This paper is based on results obtained from a project, JPNP16007, commissioned by the New Energy and Industrial Technology Development Organization (NEDO). We thank Dr. Yukoh Matsumoto (TOPS Systems Corp.) for a great help in porting the TPE to FPGA. We also thank the anonymous reviewers for their many insightful comments and suggestions.

References

[1] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," 3rd Int. Conf. Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings, ed. Y. Bengio and Y. LeCun, 2015.

[2] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.Y. Fu, and A.C. Berg, "SSD: Single Shot MultiBox Detector," Lect. Notes Comput. Sci., p.21–37, 2016.

[3] WikiChip Fuse, "Inside Tesla's neural processor in the FSD chip," https://fuse.wikichip.org/news/2707/inside-teslas-neural-processor-in-the-fsd-chip/, 2019.

[4] P.S. Labini, M. Cianfriglia, D. Perri, O. Gervasi, G. Fursin, A. Lokhmotov, C. Nugteren, B. Carpentieri, F. Zollo, and F. Vella, "On the anatomy of predictive models for accelerating GPU convolution kernels and beyond," ACM Trans. Archit. Code Optim., vol.18, no.1, Jan. 2021.

[5] J.J. Dongarra, J. Du Croz, S. Hammarling, and I.S. Duff, "A set of level 3 basic linear algebra subprograms," ACM Trans. Math. Softw., vol.16, no.1, pp.1–17, March 1990.

[6] P. Warden, "Why GEMM is at the heart of deep learning," https://shorturl.at/htQW8, 2015.

[7] A. Vasudevan, A. Anderson, and D. Gregg, "Parallel multi channel convolution using general matrix multiplication," 2017 IEEE 28th Int. Conf. Application-specific Systems, Architectures and Processors (ASAP), pp.19–24, 2017.

[8] H. Kung and C. Leiserson, "Algorithms for VLSI processor arrays," in Introduction to VLSI Systems, ed. C. Mead and L. Conway, ch. 8, pp.271–292, Addison-Wesley, Reading, MA, 1980.

[9] H.T. Kung, "Why Systolic Architectures?," Computer, vol.15, no.1, pp.37–46, Jan. 1982.

[10] N.P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T.V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. Richard Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D.H. Yoon, "In-datacenter performance analysis of a tensor processing unit," Proc. 44th Annual Int. Symp. Computer Architecture, ISCA '17, pp.1–12, ACM, New York, NY, USA, June 2017.

[11] Google's Cloud TPUs, "System architecture," https://shorturl.at/huINS, 2019.

[12] N.P. Jouppi, D.H. Yoon, G. Kurian, S. Li, N. Patil, J. Laudon, C. Young, and D. Patterson, "A domain-specific supercomputer for training deep neural networks," Commun. ACM, vol.63, no.7, pp.67–78, June 2020.

[13] N.P. Jouppi, D.H. Yoon, M. Ashcraft, M. Gottscho, T.B. Jablin, G. Kurian, J. Laudon, S. Li, P. Ma, X. Ma, T. Norrie, N. Patil, S. Prasad, C. Young, Z. Zhou, and D. Patterson, "Ten lessons from three generations shaped Google's TPUv4i: Industrial product," 2021 ACM/IEEE 48th Annual Int. Symp. Computer Architecture (ISCA), pp.1–14, IEEE Computer Society, Los Alamitos, CA, USA, June 2021.

[14] H. Genc, A. Haj-Ali, V. Iyer, A. Amid, H. Mao, J. Wright, C. Schmidt, J. Zhao, A.J. Ou, M. Banister, Y.S. Shao, B. Nikolic, I. Stoica, and K. Asanovic, "Gemmini: An agile systolic array generator enabling systematic evaluations of deep-learning architectures," CoRR, vol.abs/1911.09925, 2019.

[15] C. Mead and L. Conway, "Introduction to VLSI systems," Reading, MA, Addison-Wesley Publishing Co., 1980. 426 p., vol.-1, Jan. 1980.

[16] S. Sedukhin, "Design and analysis of systolic algorithms and structures," Programming and Computer Software, vol.17, no.2, pp.73–88, 1992.

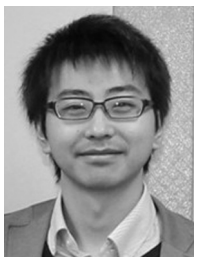
[17] R.C. Agarwal, F.G. Gustavson, and M. Zubair, "A high-performance matrix-multiplication algorithm on a distributed-memory parallel computer, using overlapped communication," IBM J. Res. Dev., vol.38, no.6, pp.673–681, Nov. 1994.

[18] R.A. van de Geijn and J. Watts, "SUMMA: Scalable universal matrix multiplication algorithm," Tech. Rep., University of Texas at Austin, USA, TR-95-13, 1995.

[19] E. Talpes, D.D. Sarma, G. Venkataramanan, P. Bannon, B. McGee, B. Floering, A. Jalote, C. Hsiong, S. Arora, A. Gorti, and G.S.

Sachdev, "Compute solution for Tesla's full self-driving computer," IEEE Micro, vol.40, no.2, pp.25–35, March-April 2020.
[20] H. Kwon, P. Chatarasi, M. Pellauer, A. Parashar, V. Sarkar, and T. Krishna, "Understanding reuse, performance, and hardware cost of DNN dataflow: A data-centric approach," Proc. 52nd Annual IEEE/ACM Int. Symp. Microarchitecture, MICRO '52, pp.754–768, Association for Computing Machinery, New York, NY, USA, 2019.
[21] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," 2016 IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR), pp.770–778, 2016.
[22] T. Liu, M. Chen, M. Zhou, S.S. Du, E. Zhou, and T. Zhao, "Towards understanding the importance of shortcut connections in residual networks," 2019.
[23] M. Horowitz, "Computing's energy problem (and what we can do about it)," 2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC), pp.10–14, 2014.
[24] B. Murmann, "Mixed-signal computing for deep neural network inference," IEEE Trans. Very Large Scale Integr. (VLSI) Syst., vol.29, no.1, pp.3–13, Jan. 2021.
[25] M.H. Sunwoo and J.K. Aggarwal, "A sliding memory plane array processor," IEEE Trans. Parallel Distrib. Syst., vol.4, no.6, pp.601–612, June 1993.
[26] N. Li, Y. Tomioka, and H. Kitazawa, "An FPGA implementation of deep convolutional neural network using synchronous shift data transfer," Tech. Rep. 426, IEICE Technical Report (VLD2014-140), Jan. 2015.
[27] S. Sedukhin, K. Matsumoto, and Y. Tomioka, "Brain-inspired co-design of algorithm/architecture for CNN accelerators," 8th International Congress on Advanced Applied Informatics, IIAI-AAI 2019, Toyama, Japan, July 7-11, 2019, pp.556–560, IEEE, 2019.
[28] G. Li, F. Li, T. Zhao, and J. Cheng, "Block convolution: Towards memory-efficient inference of large-scale cnns on FPGA," 2018 Design, Automation & Test in Europe Conference & Exhibition, DATE 2018, Dresden, Germany, March 19-23, 2018, ed. J. Madsen and A.K. Coskun, pp.1163–1166, IEEE, 2018.
[29] Blog, "Iteratively load image block-by-block where blocks are partially overlapped," https://shorturl.at/hvDW3, 2019. [Online; accessed 12-June-2021].
[30] S. Sedukhin, Y. Tomioka, and K. Yamamoto, "In search of the performance- and energy-efficient CNN accelerators," 2021 IEEE Symposium in Low-Power and High-Speed Chips (COOL CHIPS), pp.1–6, IEEE Computer Society, Los Alamitos, CA, USA, April 2021.
[31] V. Sze, Y. Chen, T. Yang, and J.S. Emer, Efficient Processing of Deep Neural Networks, Synthesis Lectures on Computer Architecture, Morgan & Claypool Publishers, 2020.
[32] X. Ding, X. Zhang, N. Ma, J. Han, G. Ding, and J. Sun, "RepVGG: Making VGG-style ConvNets great again," CoRR, vol.abs/2101.03697, 2021.
[33] S. Alyamkin, M. Ardi, A. Brighton, A.C. Berg, Y. Chen, H.-P. Cheng, B. Chen, Z. Fan, C. Feng, B. Fu, K. Gauen, J. Go, A. Goncharenko, X. Guo, H.H. Nguyen, A. Howard, Y. Huang, D. Kang, J. Kim, A. Kondratyev, S. Lee, S. Lee, J. Lee, Z. Liang, X. Liu, J. Liu, Z. Li, Y. Lu, Y.-H. Lu, D. Malik, E. Park, D. Repin, T. Sheng, L. Shen, F. Sun, D. Svitov, G.K. Thiruvathukal, B. Zhang, J. Zhang, X. Zhang, and S. Zhuo, "2018 low-power image recognition challenge," CoRR, vol.abs/1810.01732, 2018.
[34] N. Shazeer, K. Fatahalian, W.R. Mark, and R.T. Mullapudi, "Hydranets: Specialized dynamic architectures for efficient inference," 2018 IEEE/CVF Conf. Comput. Vis. Pattern Recognit., pp.8080–8089, 2018.
[35] T. Garipov, D. Podoprikhin, A. Novikov, and D.P. Vetrov, "Ultimate tensorization: compressing convolutional and FC layers alike," http://arxiv.org/abs/1611.03214, 2016.
[36] D.I. Moldovan, "On the design of algorithms for vlsi systolic arrays," Proc. IEEE, vol.71, no.1, pp.113–120, Jan. 1983.
[37] P. Quinton, "Automatic synthesis of systolic arrays from uniform recurrent equations," SIGARCH Comput. Archit. News, vol.12, no.3, pp.208–214, Jan. 1984.
[38] P.R. Cappello and K. Steiglitz, "Selecting systolic designs using linear transformations of space-time," Real-Time Signal Processing VII, ed. K. Bromley, pp.75–85, International Society for Optics and Photonics, SPIE, 1984.
[39] S. Kung, "VLSI array processors," IEEE ASSP Magazine, vol.2, no.3, pp.4–22, July 1985.
[40] S.G. Sedukhin, A.S. Zekri, and T. Myiazaki, "Orbital algorithms and unified array processor for computing 2d separable transforms," 2010 39th Int. Conf. Parallel Processing Workshops, pp.127–134, 2010.
[41] X. Zhang, J. Zou, K. He, and J. Sun, "Accelerating very deep convolutional networks for classification and detection," IEEE Trans. Pattern Anal. Mach. Intell., vol.38, no.10, pp.1943–1955, Oct. 2016.
[42] J. Redmon and A. Farhadi, "YOLOv3: An incremental improvement," http://arxiv.org/abs/1804.02767, 2018.
[43] K. Yamamoto and K. Maeno, "PCAS: pruning channels with attention statistics for deep network compression," 30th British Machine Vision Conference 2019, BMVC 2019, Cardiff, UK, Sept. 9-12, 2019, p.138, BMVA Press, 2019.
[44] Y. 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.
[45] A. Bytyn, R. Leupers, and G. Ascheid, "An application-specific VLIW processor with vector instruction set for CNN acceleration," IEEE Int. Symp. Circuits and Systems, ISCAS 2019, Sapporo, Japan, May 26-29, 2019, pp.1–5, IEEE, 2019.
[46] T. Norrie, N. Patil, D.H. Yoon, G. Kurian, S. Li, J. Laudon, C. Young, N. Jouppi, and D. Patterson, "The design process for Google's training chips: TPUv2 and TPUv3," IEEE Micro, vol.41, no.2, pp.56–63, March-April 2021.
[47] M. Alwani, H. Chen, M. Ferdman, and P. Milder, "Fused-layer CNN accelerators," 2016 49th Annual IEEE/ACM Int. Symp. Microarchitecture (MICRO), pp.1–12, 2016.
[48] P. Seitz, "Smart image sensors: an emerging key technology for advanced optical measurement and microsystems," Micro-Optical Technologies for Measurement, Sensors, and Microsystems, ed. O.M. Parriaux, pp.244–255, International Society for Optics and Photonics, SPIE, 1996.
[49] E.H.M. Heijne, "Gigasensors for an attoscope: Catching quanta in CMOS," IEEE Solid-State Circuits Society Newsletter, vol.13, no.4, pp.28–34, 2008.

Stanislav Sedukhin received his Ph.D. and Dr.Sci. (habilitation) from the Russian Academy of Sciences in 1982 and 1993, respectively. From 1993 he joined the University of Aizu where he was a professor, Dean of Graduate School and Vice-President. Dr. Sedukhin research interests are in architectural synthesis of application-specific array processors and massively-parallel algorithms. Currently, he is Professor Emeritus of the University of Aizu.



Yoichi Tomioka received the B.E., M.E., and D.E. degrees from the Tokyo Institute of Technology, Tokyo, Japan, in 2005, 2006, and 2009, respectively. He was a Research Associate with the Tokyo Institute of Technology until 2009. He was an Assistant Professor with the Division of Advanced Electrical and Electronics Engineering, Tokyo University of Agriculture and Technology until 2015. He was an Associate Professor with the School of Computer Science and Engineering, University of Aizu until 2018. Since 2019, he has been an Senior Associate Professor with the university. His research interests include image processing, hardware acceleration, high-performance computing, electrical design automation, and combinational algorithms. He is a member of IPSJ and IEEE.

Kohei Yamamoto received his B.E., and M.E. degrees from Department of Management Systems Engineering, Chuo University, Tokyo, Japan, in 2012 and 2014, respectively. He is currently a research engineer at Oki Electric Industry Co., Ltd. working on developing machine learning algorithms.