Deep Convolutional Neural Network Architecture With Reconfigurable Computation Patterns

Fengbin Tu, Shouyi Yin, *Member, IEEE*, Peng Ouyang, Shibin Tang,
Leibo Liu, *Member, IEEE*, and Shaojun Wei, *Member, IEEE*

***Abstract*— Deep convolutional neural networks (DCNNs) have been successfully used in many computer vision tasks. Previous works on DCNN acceleration usually use a fixed computation pattern for diverse DCNN models, leading to imbalance between power efficiency and performance. We solve this problem by designing a DCNN acceleration architecture called deep neural architecture (DNA), with reconfigurable computation patterns for different models. The computation pattern comprises a data reuse pattern and a convolution mapping method. For massive and different layer sizes, DNA reconfigures its data paths to support a hybrid data reuse pattern, which reduces total energy consumption by 5.9∼8.4 times over conventional methods. For various convolution parameters, DNA reconfigures its computing resources to support a highly scalable convolution mapping method, which obtains 93% computing resource utilization on modern DCNNs. Finally, a layer-based scheduling framework is proposed to balance DNA's power efficiency and performance for different DCNNs. DNA is implemented in the area of 16 mm^2 at 65 nm. On the benchmarks, it achieves 194.4 GOPS at 200 MHz and consumes only 479 mW. The system-level power efficiency is 152.9 GOPS/W (considering DRAM access power), which outperforms the state-of-the-art designs by one to two orders.**

***Index Terms*— Computation pattern, deep convolutional neural networks (DCNNs), neural network accelerator, reconfigurable architecture, scheduling framework.**

I. Introduction

DEEP CONVOLUTIONAL NEURAL NETWORKS (DCNNs), as shown in Fig. 1, have been widely used in various computer vision applications, such as image classification, object detection, and video surveillance. However, the massive data movements and computational complexity of DCNNs raise big challenges to power efficiency and performance, which hinders DCNN's application in embedded devices, such as smartphones and intelligent vehicles. Currently, DCNNs are often accelerated on general-purpose platforms, such as CPU or GPU [1], [2]. However, performance on CPU cannot meet the real-time processing requirements in embedded devices [3]. GPU's power consumption is too high for embedded computing (235 W for NVIDIA Tesla K40 GPU [4]). field-programmable gate array (FPGA) shows acceptable performance and power, but its fine-grained computing and routing resources still limit the power efficiency and runtime reconfiguration for different DCNNs [5].

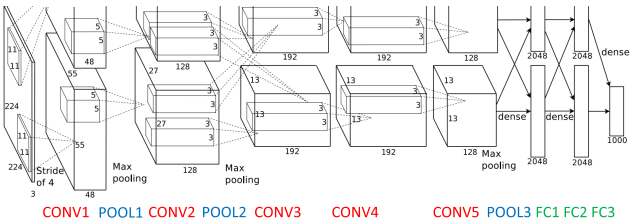


Fig. 1. AlexNet: a typical DCNN model [6].

Many efficient DCNN accelerators have been designed with specialized processing elements (PEs) and data paths [7], [8]. These works usually use fixed data reuse patterns for different DCNN models. For example, [7] has the minimum access to their output storage, while Chen *et al.*'s [8] pattern minimizes input access, leading to different power efficiencies and performances. Due to the diversity of DCNN layer sizes, a fixed data reuse pattern may not fit well when the model changes, which would increase data movements and harm power efficiency. Moreover, their convolution mapping method is not very scalable for various convolution parameters. Mismatch arises between network shapes and computing resources, which lowers resource utilization and performance.

Therefore, a DCNN accelerating architecture should have reconfigurability for different networks and consider optimizations for both power efficiency and performance. To realize this goal, we design a reconfigurable architecture called deep neural architecture (DNA), with reconfigurable computation patterns for different DCNNs. This paper outperforms the previous works for three reasons.

1) DNA can reconfigure its data paths to support a hybrid data reuse pattern for different layer sizes, instead of the fixed styles in [5], [7], and [8] (see Sections III-B and IV-A).
2) DNA can reconfigure its computing resources to support a highly scalable and efficient mapping method, thus improving resource utilization for different convolution parameters (see Sections III-C and IV-B).
3) A layer-based scheduling framework is proposed to reconfigure DNA's resources with optimization on both power efficiency and performance (see Section IV-C).

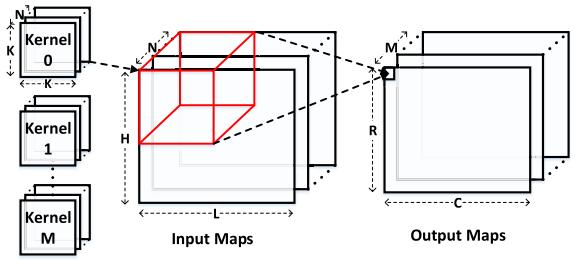
In this paper, we start from algorithm-level optimizations for DCNNs, and then implement a reconfigurable architecture

Manuscript received October 11, 2016; revised February 5, 2017; accepted March 19, 2017. Date of publication April 12, 2017; date of current version July 24, 2017. This work was supported in part by the China Major S&T Project under Grant 2013ZX01033001-001-003, in part by the National High-Tech R&D Program of China under Grant 2015AA016601, and in part by the NNSF of China under Grant 61274131. *(Corresponding author: Shouyi Yin.)*

The authors are with the National Laboratory for Information Science and Technology, Institute of Microelectronics, Tsinghua University, Beijing 100084, China (e-mail: yinsy@tsinghua.edu.cn).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TVLSI.2017.2688340



(a)

```
for(r = 0; r < R; r++)                    // Loop R.
  for(c = 0; c < C; c++)                  // Loop C.
    for(m = 0; m < M; m++)                // Loop M.
      for(n = 0; n < N; n++)              // Loop N.
        O[m][r][c] +=                     // Convolution.
          Σ_{i=0}^{K-1} Σ_{j=0}^{K-1} W[m][n][i][j] * I[n][r*S+i][c*S+j];
```

(b)

Fig. 2. Illustration for a CONV layer. (a) CONV layer in DCNNs. (b) Pseudocode of a CONV layer.

to support the optimized computation pattern for each layer. The rest of this paper is organized as follows. Section II provides a preliminary on DCNN and a basic architecture for DCNN acceleration. Section III gives a detailed discussion on data reuse patterns and proposes a scalable mapping strategy. Section IV describes the reconfigurable architecture design and work flow. Section V presents the experimental results. Section VI summarizes this paper.

II. Preliminary

A. DCNN Model

Fig. 1 shows a typical DCNN model called AlexNet [6], which classifies a 224×224 RGB image into 1000 categories. Like AlexNet, most DCNNs are composed of three types of layers: convolutional (CONV) layers, pooling (POOL) layers, and full connection (FC) layers. CONV layers extract different levels of features from the input image. POOL layers follow CONV layers and perform subsampling onto the feature maps. FC layers are usually used in the last few layers and act as a classifier. Since CONV layers account for more than 90% of DCNN's runtime [3], our discussion is focused on accelerating the CONV layers of DCNNs. The proposed method can be easily extended to other layers.

Fig. 2(a) shows a CONV layer in DCNNs. It takes $N \times H \times L$ feature maps as the inputs, and has M 3-D convolutional kernels ($K \times K \times N$). Each kernel performs a 3-D convolution on the input maps with a sliding stride of S, which generates an $R \times C$ output map. Therefore, the output map number equals to the kernel number M. The computation of a CONV layer can be expressed as the multilayer loop in Fig. 2(b), where matrices I, O, and W stand for the input maps, output maps, and kernel weights.

Take AlexNet as an example, we find out that a DCNN's CONV layers have various kernel sizes, map numbers, and convolution strides, for different levels of feature extraction (see Table I). The variety leads to different demands for computing and storage resources. The last four rows of Table I list the total number of inputs (TI), outputs (TO), weights (TW), and multiply accumulate (MAC) operations (TM). We can see that the first layer (CONV1) has the largest number of input/output data and MAC operations. For the rest layers, although the data size and MAC number shrink, the total of kernel weights increases. Moreover, in order to achieve higher accuracy, DCNNs with hundreds of layers have been developed [9].

TABLE I
Parameters of AlexNet's CONV Layers

CONV Layer	1	2	3	4	5
Imap Size (H, L)	224	27	13	13	13
Imap Number (N)	3	48	256	192	192
Imap Group (G)	1	2	1	2	2
Omap Size (R, C)	55	27	13	13	13
Omap Number (M)	96	256	384	384	256
Kernel Size (K)	11	5	3	3	3
Stride (S)	4	1	1	1	1
Total Inputs (TI)	151k	70k	43k	65k	65k
Total Outputs (TO)	290k	187k	65k	65k	43k
Total Weights (TW)	35k	307k	885k	664k	442k
Total MACs (TM)	35138k	4666k	584k	584k	389k

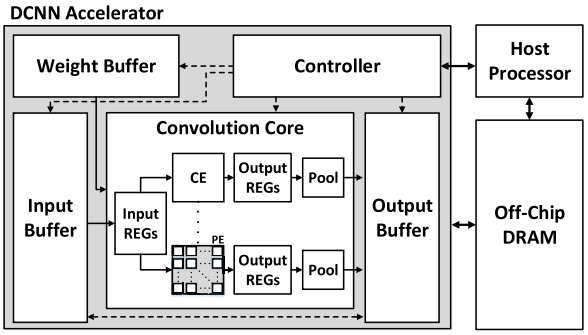


Fig. 3. DCNN accelerating system consisting of a DCNN accelerator, an off-chip DRAM, and a host processor.

B. Architecture of a DCNN Accelerating System

The large number of layers in DCNNs makes it difficult to map the entire network onto a single chip. Instead, most designs use a layer-by-layer accelerating style [5], [7], [8]. The layers execute sequentially, and the current outputs are fed to the next layer as inputs. Fig. 3 shows a typical DCNN accelerating system used in many works, such as [5], [7], and [8]. The system consists of a DCNN accelerator, an off-chip DRAM and a host processor. In the DCNN accelerator, there is a controller, weight/input/output buffers and a convolution core. The controller configures the other blocks with different layer parameters. The input buffer and weight buffer store the input maps and kernel weights of the current layer. The convolution core loads feature maps to its input registers (Input REGs) and then performs convolutions with several parallel convolution engines (CEs), which are composed of many PEs. The output maps stored in CEs'

output registers (Output REGs) are sent to the output buffer through pooling units. Once the current layer is completed, the outputs are switched to the inputs or stored in the off-chip DRAM for the next layer, to realize the layer-based style. All the discussions in this paper are based on the typical architecture.

C. Two Challenges to DCNN Acceleration

Power efficiency and performance are both important in DCNN acceleration. Power efficiency is defined as the operation count per second per watt (OPS/W)

$$\text{Power Efficiency} = \frac{\text{Performance}}{\text{Power}} = \frac{\text{Operations/Time}}{\text{Energy/Time}}. \quad (1)$$

For a certain DCNN, its total number of operations (multiply accumulation) is constant, so power efficiency is $\propto$ (1/Energy). Energy consumption is the sum of total memory access energy and computation energy

$$\begin{aligned}\text{Energy} = {} & \text{MA}_{\text{DRAM}} \cdot E_{\text{DRAM}} + \text{MA}_{\text{buffer}} \cdot E_{\text{buffer}} \\ & + \text{Operations} \cdot E_{\text{operation}} \end{aligned} \quad (2)$$

where E_{DRAM}, E_{buffer}, and $E_{\text{operation}}$ are the energy consumption for each DRAM access, buffer access, and each operation, respectively. The third term is also constant for a certain network, so power efficiency is directly determined by total DRAM access and buffer access times during execution. However, the variety of layer sizes becomes a big challenge.

Challenge 1: The variety of layer sizes demands that we should efficiently reuse different layer's data in on-chip buffers. Otherwise, more memory access would be caused, which lowers power efficiency.

Performance of DCNN acceleration can be defined as

$$\text{Performance} = 2\text{MAC} \cdot \text{PE Utilization} \cdot \text{Frequency} \quad (3)$$

where MAC is the total number of multiply accumulators contained in all PEs of the accelerator, and each MAC operation has two basic operations. So the performance is decided by the accelerator's PE utilization on the given network. However, diverse convolution parameters, such as strides, kernel sizes, and feature map sizes, could produce mismatch with the accelerator's fixed computing structure and then lower the performance, which is another challenge.

Challenge 2: The accelerator should make good use of the available PEs to support different convolution parameters and improve its performance.

III. Computation Pattern

A. Overview

For the two challenges that obstacle efficient DCNN acceleration, we will discuss how its computation pattern can be optimized from the algorithm level. The proposed method would guide designing the architecture in Section IV. It is considered in the data reuse and convolution mapping aspects. The massive and diverse layer sizes result in substantial and varying data movements among the convolution core, buffers, and off-chip DRAM. A proper data reuse pattern would reduce redundant memory access to save energy and improve power efficiency. Convolution mapping describes how convolution kernels are performed in the convolution core, which can be optimized for higher PE utilization under different convolution parameters.

B. Data Reuse Pattern

In this part, we propose a hybrid data reuse pattern that combines three basic reuse patterns. Memory access times are used to measure the data movements between the core and buffers based on the architecture model in Fig. 3. Due to the limited storage in the convolution core, we apply tiling to the CONV layer. As shown in Fig. 5(a), the input and output feature maps are tiled by Th $\times$ Tl $\times$ Tn and Tr $\times$ Tc $\times$ Tm. Th and Tl can be computed by $\text{Th} = (\text{Tr} - 1)S + K$ and $\text{Tl} = (\text{Tc} - 1)S + K$, where K and S are the kernel size and stride. For both buffer access and DRAM access, the total access times can be expressed in a uniform equation

$$\text{MA} = \text{TI} \cdot \alpha_i + \text{TO} \cdot \alpha_o + \text{TW} \cdot \alpha_w + \text{TPO} \quad (4)$$

where TI, TO, and TW stand for the total number of inputs, outputs, and kernel weights, and α_i, α_o, and α_w are their reuse times during computation. TPO is short for total pooled outputs, which describes the number of data written to the memory after pooling, and usually equals to TO/4.

1) Input Reuse: As shown in Fig. 5(a), we first introduce one basic data reuse pattern with minimum access to the input buffer, called input reuse (IR). Similar to Chen *et al.*'s [8] method, IR has three stages: 1) the core loads input maps to the Input REGs; 2) those input maps are fully reused to update all corresponding partial sums stored in the output buffer; and 3) the partial sums are sent back to the output buffer through bypassing the pooling units. They will be fetched by the core to continue the accumulation in depth when the next input map is loaded.

In IR, the CONV layer is equivalently transformed as the pseudocode in Fig. 5(b), where the four outermost loops indicate the data reuse pattern and the innermost four loops represent the computation in the core. In IR, Loop N is the outer loop of Loop M, meaning each input map is fully reused for computation before loading the next map. The core's total buffer access in IR is $\text{MA}_{\text{buffer}}^{(\text{IR})}$

$$\alpha_i = 1, \quad \alpha_o = 2\left(\left\lceil \frac{N}{\text{Tn}} \right\rceil - 1\right), \quad \alpha_w = \left\lceil \frac{H}{\text{Th}} \right\rceil \left\lceil \frac{L}{\text{Tl}} \right\rceil \quad (5)$$

which follows (4). In IR, all the input maps are loaded only once, so α_i equals to 1. Note that TI is enlarged by 1$\sim$1.5 times over the original size, because the overlapping part between two tiles nearby is reloaded from the input buffer. As the example provided in Fig. 4, we tile the 24 $\times$ 24 output map into nine nonoverlapping 8 $\times$ 8 tiles (kernel size $K = 3$ and stride $S = 1$). Thus, in the input map, there exists overlapping regions of 10 $\times$ 2 [see Fig. 4 (gray)] between neighboring tiles. In our method, the input tiles are separately loaded from memory without reusing the overlapping data. For Input Tile[i, j], we load it from its top-left corner, at [$i \cdot (\text{Th} - K + 1)$, $j \cdot (\text{Tl} - K + 1)$]. In this case, each input map is enlarged from 26$\times$26 to 30$\times$30, increasing TI by 1.33 times.

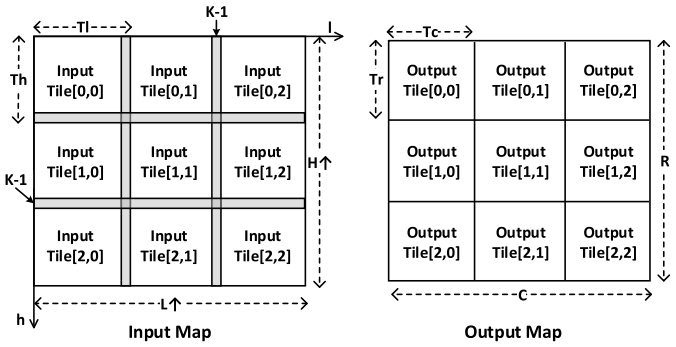


Fig. 4. Illustration for the overlapping regions caused by tiling ($H = L = 26 \rightarrow 30$, $\mathrm{Th} = \mathrm{Tl} = 10$, $R = C = 24$, $\mathrm{Tr} = \mathrm{Tc} = 8$, $K = 3$, and $S = 1$).

Since the Input REGs store Tn maps, the core needs to read and write the partial sums for $\lceil (N/\mathrm{Tn}) \rceil - 1$ times, to accumulate in depth for 3-D convolutions. Thus, α_o is $2(\lceil (N/\mathrm{Tn}) \rceil - 1)$. As for weight loading, the TW weights should be loaded for each input tile, so the last term is $\mathrm{TW} \cdot \lceil (H/\mathrm{Th}) \rceil \lceil (L/\mathrm{Tl}) \rceil$. H and L are also enlarged due to the reloading of overlapping data.

If the cached data in IR exceed the buffer size, extra DRAM access would be incurred. So the accelerator's total DRAM access $\mathrm{MA}_{\mathrm{DRAM}}^{(\mathrm{IR})}$ can be expressed as follows:

$$\alpha_i = 1, \quad \alpha_o = \begin{cases} 0, & M \cdot \mathrm{Tr} \cdot \mathrm{Tc} \le B_o \\ 2\left(\left\lceil \frac{N}{\mathrm{Tn}} \right\rceil - 1\right), & M \cdot \mathrm{Tr} \cdot \mathrm{Tc} > B_o \end{cases}$$
$$\alpha_w = \begin{cases} 1, & \mathrm{TW} \le B_w \\ \left\lceil \frac{H}{\mathrm{Th}} \right\rceil \left\lceil \frac{L}{\mathrm{Tl}} \right\rceil, & \mathrm{TW} > B_w \end{cases} \tag{6}$$

where B_o and B_w refer to the capacity of the output buffer and weight buffer. The input maps are still loaded only once, while the output and weight access times depend on the cached data size. If we can cache $M \cdot \mathrm{Tr} \cdot \mathrm{Tc}$ data in the output buffer ($M \cdot \mathrm{Tr} \cdot \mathrm{Tc} \le B_o$), no partial sums need to write into DRAM during computation ($\alpha_o = 0$). Otherwise, the partial sums should be stored in DRAM, leading to repeated DRAM access ($\alpha_o = 2(\lceil (N/\mathrm{Tn}) \rceil - 1)$). Similarly, the weight access times to DRAM can be minimized if the cached data does not exceed the weight buffer size B_w. Since Loops R and C are the outermost two loops in Fig. 5(b), we need to cache all the weights (TW) to reduce the DRAM access.

2) Output Reuse: Considering memory access to the output buffer, we introduce another basic pattern called output reuse (OR), as shown in Fig. 5(c). Similar to the reuse patterns of [5] and [7], OR also has three stages: 1) different channels of the same input tile are successively loaded into the core; 2) the partial sums in the Output REGs are reused until the tiled output map is completed; and 3) the final output maps are stored after pooling, so there is no other access to the output buffer during computation.

The CONV layer is equivalently transformed into the loop in Fig. 5(d). Compared with the loop in Fig. 5(b), Loops M and Loop N are interchanged, so the maps are sent to the output buffer after finishing all the accumulations. Therefore, the total buffer access in OR is $\mathrm{MA}_{\mathrm{buffer}}^{(\mathrm{OR})}$

$$\alpha_i = \left\lceil \frac{M}{\mathrm{Tm}} \right\rceil, \quad \alpha_o = 0, \quad \alpha_w = \left\lceil \frac{R}{\mathrm{Tr}} \right\rceil \left\lceil \frac{C}{\mathrm{Tc}} \right\rceil. \tag{7}$$

Since the convolution core can store Tm maps in its Output REGs, each input map should be loaded $\lceil (M/\mathrm{Tm}) \rceil$ times. Different from IR, the kernel weights are reloaded for each output tile in OR, so α_w is equal to $\lceil (R/\mathrm{Tr}) \rceil \lceil (C/\mathrm{Tc}) \rceil$.

In OR, the DRAM access $\mathrm{MA}_{\mathrm{DRAM}}^{(\mathrm{OR})}$ is also strongly related to $\mathrm{MA}_{\mathrm{buffer}}^{(\mathrm{OR})}$ and buffer sizes

$$\alpha_i = \begin{cases} 1, & N \cdot \mathrm{Th} \cdot \mathrm{Tl} \le B_i \\ \left\lceil \frac{M}{\mathrm{Tm}} \right\rceil, & N \cdot \mathrm{Th} \cdot \mathrm{Tl} > B_i \end{cases}, \quad \alpha_o = 0$$
$$\alpha_w = \begin{cases} 1, & \mathrm{TW} \le B_w \\ \left\lceil \frac{R}{\mathrm{Tr}} \right\rceil \left\lceil \frac{C}{\mathrm{Tc}} \right\rceil, & \mathrm{TW} > B_w \end{cases} \tag{8}$$

where B_i refers to the input buffer's capacity. If the input buffer can store N input tiles ($N \cdot \mathrm{Th} \cdot \mathrm{Tl} \le B_i$), the inputs need to load only once ($\alpha_i = 1$).

3) Weight Reuse: In IR and OR, repeated weight access to DRAM would occur if TW exceeds the weight buffer size. So we propose a reuse pattern named weight reuse (WR), to minimize DRAM access for weights. Shown in Fig. 5(e), WR has four stages: 1) the core loads Tn tiled input maps to the Input REGs; 2) those input maps are used to update Tm corresponding partial sums; 3) the $\mathrm{Tn} \times \mathrm{Tm}$ kernel weights in the weight buffer are fully reused to compute Tm maps of $R \times C$; and 5) the partial sums are fetched to complete the convolution with the rest inputs and weights.

In WR, the CONV layer is transformed to the pseudocode in Fig. 5(f). Compared with IR and OR, Loop M and Loop N are switched to the outermost loops to make better use of kernel weights. WR's buffer access and DRAM access are as follows.

1) $\mathrm{MA}_{\mathrm{buffer}}^{(\mathrm{WR})}$:

$$\alpha_i = \left\lceil \frac{M}{\mathrm{Tm}} \right\rceil, \quad \alpha_o = 2\left(\left\lceil \frac{N}{\mathrm{Tn}} \right\rceil - 1\right)$$
$$\alpha_w = \left\lceil \frac{R}{\mathrm{Tr}} \right\rceil \left\lceil \frac{C}{\mathrm{Tc}} \right\rceil. \tag{9}$$

2) $\mathrm{MA}_{\mathrm{DRAM}}^{(\mathrm{WR})}$:

$$\alpha_i = \begin{cases} 1, & \mathrm{TI} \le B_i \\ \left\lceil \frac{M}{\mathrm{Tm}} \right\rceil, & \mathrm{TI} > B_i. \end{cases}$$
$$\alpha_o = \begin{cases} 0, & \mathrm{Tm} \cdot R \cdot C \le B_o \\ 2\left(\left\lceil \frac{N}{\mathrm{Tn}} \right\rceil - 1\right), & \mathrm{Tm} \cdot R \cdot C > B_o \end{cases}$$
$$\alpha_w = 1. \tag{10}$$

Through fully reusing kernel weights in the buffer, we only need to load weights from DRAM for only once.

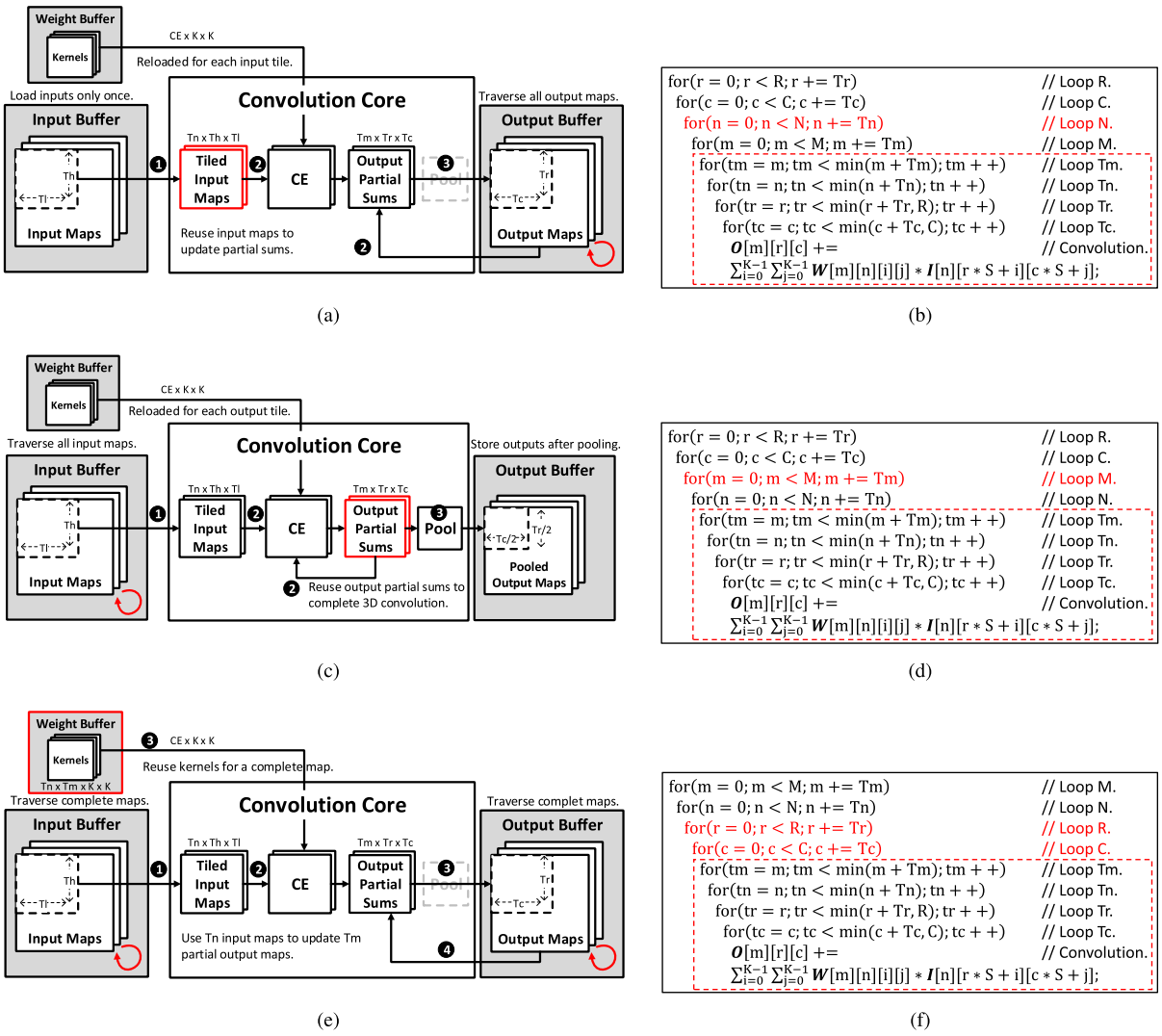


Fig. 5. Data reuse patterns: IR, OR, and WR. (a), (c), and (e) illustrate the stages of them. (b), (d), and (f) present their equivalent loop transformation. The four outermost loops indicate their difference in data movements. The innermost four loops (in the red box) refer to the computation in the convolution core, with tiling parameters of ⟨Tm, Tn, Tr, Tc⟩. Illustration for (a) IR, (c) OR, and (e) WR. Pseudocode of (b) IR, (d) OR, and (f) WR.

4) Hybrid Data Reuse: As shown in Fig. 6, we analyze IR, OR, and WR's buffer access times on CONV3 of AlexNet by sweeping Tr and Tc from 2 to 14. We set the tiling parameters limited by $Tn \cdot Th \cdot Tl \leq 40^2$ and $Tm \cdot Tr \cdot Tc \leq 32^2$, corresponding to our exact implementation in Section IV. The growing of Tr × Tc leads to larger feature maps and more intramap data reuse. But the input/output map number Tm and Tn decreases, reducing intermap data reuse. Thus, increasing Tr × Tc leads to different memory access times for IR, OR, and WR. They achieve the minimum memory access times at different points: 12×12 for IR, 14 × 14 for OR, and 8 × 8 for WR. Meanwhile, the ratios $MA_{buffer}^{(IR)}/MA_{buffer}^{(OR)}$ and $MA_{buffer}^{(WR)}/MA_{buffer}^{(OR)}$ also vary in different cases, as shown by the yellow and blue lines in Fig. 6. The points above the dashed line indicate that IR or WR has more buffer access times than OR. As for DRAM access MA_{DRAM}, it is closely related to MA_{buffer} and the available buffer sizes. The variety would become more complicated when the layer changes. However, in previous works, they often use a fixed reuse pattern with relatively large Tr × Tc, which is not the best choice for all cases.

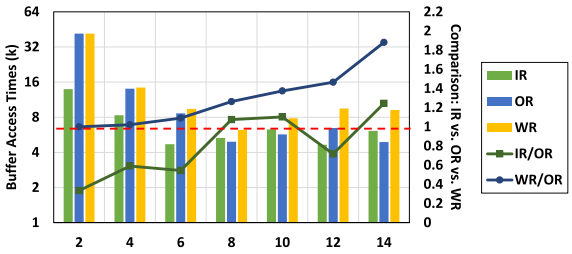


Fig. 6. Buffer access analysis on AlexNet's CONV3 layer.

Considering the variety of CONV layers, we propose a hybrid method that assigns each layer with a separate data reuse pattern, represented by ⟨IR/OR/WR, Tr, Tc, Tm, Tn⟩. We explore IR/OR/WR with different tiling parameters, and estimate each design point's memory access energy by

$$\text{Energy}_{MA} = MA_{DRAM} \cdot E_{DRAM} + MA_{buffer} \cdot E_{buffer}. \quad (11)$$

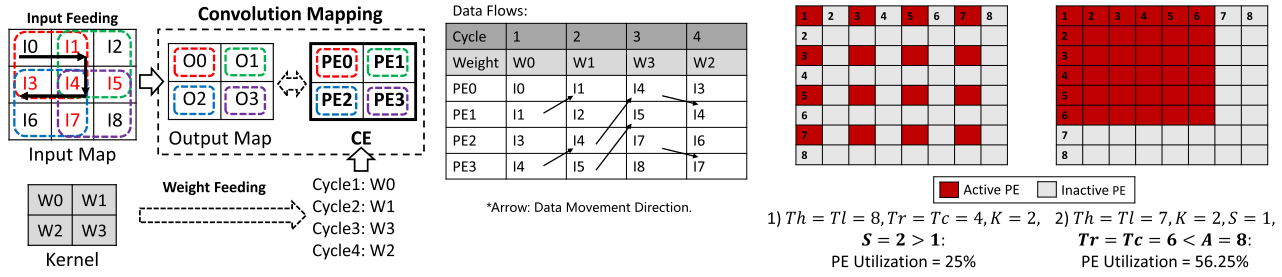


Fig. 7. Conventional convolution mapping method OOM and its limitations. (a) and (b) Running case: a 3×3 input map convolved by a 2 × 2 kernel with a stride of 1 on a 2 × 2 PE array, i.e., Th = Tl = 3, Tr = Tc = 2, $K = 2$, and $S = 1$. (c) OOM's two limitations: 1) convolution stride (S) > 1 and 2) mismatch between the irregular feature map size (Tr × Tc) and the fixed array size ($A \times A$).

We only estimate memory access energy, because MAC operation energy is constant for a certain network. In this paper, the off-chip DRAM is DDR3, whose access energy E_{DRAM} is estimated as 70 pJ/bit [10]. The buffer access energy E_{buffer} is averagely 0.42 pJ/bit in our design, extracted from the TSMC 65-nm LP technology. Based on the above-mentioned equation, the best reuse pattern is found out under the given layer size and hardware storage limitations.

C. Convolution Mapping Method

1) Output Oriented Mapping: Once an input map is loaded into the core, the next task is efficiently performing convolutions in different shapes, corresponding to the innermost four loops in Fig. 5(b), (d), and (f). To support different convolution kernel sizes and utilize data reusability during kernel sliding, previous works [5], [8], [11] usually use a convolution mapping method called output oriented mapping (OOM). In Fig. 7(a) and (b), we provide a running case to illustrate OOM: a 3 × 3 input map is convolved by a 2 × 2 kernel with a stride of 1, on a CE with 2 × 2 PEs. As shown in Fig. 7(a), the convolution kernel slides over the four colored regions to obtain the four points in the output map. Each region has $K \times K$ input data: {$I0$, $I1$, $I3$, $I4$}, {$I1$, $I2$, $I4$, $I5$}, {$I3$, $I4$, $I6$, $I7$}, and {$I4$, $I5$, $I7$, $I8$} to compute four output points $O0 \sim O3$. The principle of OOM is that the four output points are assigned to separate PEs in the CE. The PE array has interconnections to share data between neighboring PEs. So, the PEs can collect the required input data along the sliding trajectory in the input map (black arrows). The computation needs only four cycles, whose data flows are shown in Fig. 7(b).

Cycle 1: {$I0$, $I1$, $I3$, $I4$} are separately assigned to PE0 ∼ PE3 as the first input data. Meanwhile, all the PEs load the first kernel weight $W0$. At Cycle 2, the input data will be multiplied by the weights. Products will be accumulated with the previous result and stored in the PEs' output register at Cycle 3. The input loading, multiplication, and accumulation are fully pipelined.

Cycle 2: PE0 and PE2 reuse the input data previously stored in PE1 and PE3 as their second inputs ($I1$ and $I4$), as indicated by the arrows in Fig. 7(b). PE1 and PE3 load their required data ($I2$ and $I5$) from the input map. All the PEs shared the second kernel weight $W1$.

Cycle 3: All the PEs have collected their inputs in the first row, so they move down to continue the accumulation in the second row. PE0 and PE1 load $I4$ and $I5$ shifted from PE2 and PE3, while the input map feeds PE2 and PE3 with $I7$ and $I8$. Weight $W3$ is sent to all the four PEs.

Cycle 4: Different from the first row, the second row accumulation should move toward the opposite direction. Thus, PE1 and PE3 load $I4$ and $I7$ from PE0 and PE2, and $I3$ and $I6$ are sent to PE0 and PE2. The last weight $W2$ is shared by all the PEs. So far, every PE has collected its required data and weights. The final results $O0 \sim O3$ will be computed in two cycles.

2) Limitations of OOM That Lower PE Utilization: OOM is highly scalable for different kernel sizes, because PEs can easily collect their required data along the S-shaped sliding trajectory, without changing PE functions. For different kernel sizes, we just need to adjust the trajectory's length, which is equal to the kernel size ($K \times K$). However, as indicated in Fig. 7(c), there are two limitations that lower PE utilization: 1) convolution stride (S) > 1 and 2) mismatch between the irregular feature map size (Tr × Tc) and the fixed array size ($A \times A$). In Fig. 7(c-1), the convolution stride S is 2, but OOM only allows the inputs to move in a stride of 1. The data collected by the inactive PEs (in gray) are not required in $S = 2$, so the 4×4 output points scatter on the 8×8 PE array, with only 25% PE utilization. In Fig. 7(c-2), a 6×6 output map is assigned to the same array, leaving 43.75% PEs inactive during its computation.

3) Parallel Output Oriented Mapping: PE utilization of naive OOM can be defined as follows:

$$\text{PE Utilization} = \frac{R \cdot C}{A^2 \cdot \left\lceil \frac{R}{\text{Tr}} \right\rceil \left\lceil \frac{C}{\text{Tc}} \right\rceil} \tag{12}$$

where A is the PE array size. As shown in Fig. 8, we overcome OOM's limitations by parallelly computing P maps on the PE array, which is called parallel OOM (POOM). Due to the limited computing resources, increasing P would decrease the feature map size computed on the array. Thus, we perform further tiling on the innermost four loops assigned to the convolution core (see Fig. 9). In POOM, PE utilization is redefined as

$$\text{PE Utilization} = \frac{R \cdot C \cdot P}{A^2 \cdot \left\lceil \frac{R}{\text{Trr}} \right\rceil \left\lceil \frac{C}{\text{Tcc}} \right\rceil} \tag{13}$$

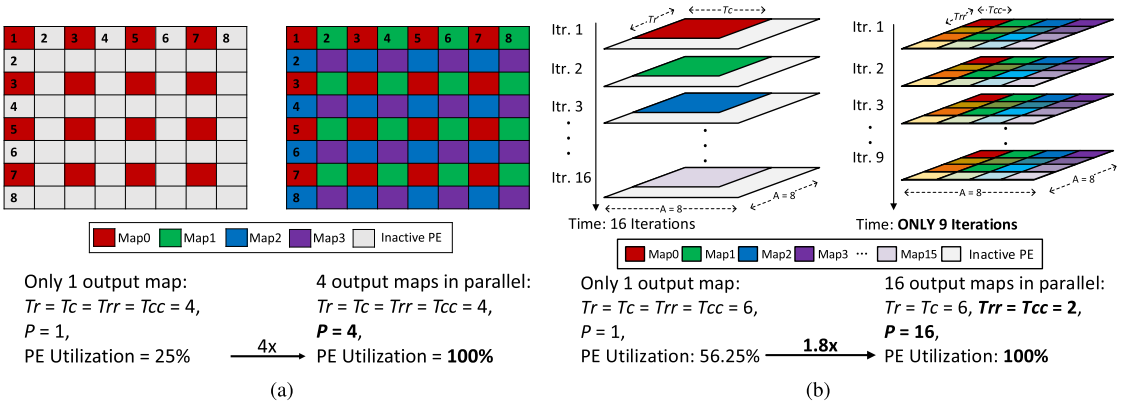


Fig. 8. POOM: improve PE utilization by paralleling multiple maps on a PE array. (a) POOM applied to a large stride case. (b) POOM applied to an irregular map size case.

where Trr and Tcc are the tiling parameters on the PE array. For the large stride ($S = 2$) case in Fig. 7(c-1), we assign four ($= S^2$) output maps in parallel to make use of the inactive PEs. They share the input data collected by the "red" PEs and compute with their own kernel weights. Thus, P rises by four times and the PE utilization is improved to 100% [see Fig. 8(a)]. POOM can also be applied to the irregular map case, as demonstrated in Fig. 8(b). In the conventional OOM, computing 16 6×6 maps need 16 iterations, with PE utilization of 56.25%. In POOM, we tile each output map into smaller blocks, so that the array can compute more maps in parallel. As the case in Fig. 8(b), the 8×8 array is divided into 16 2×2 blocks, while all blocks share the same input data. In this way, the array compute 16 2×2 tiled output maps in each iteration (Trr $\times$ Tcc : $6\times6 \rightarrow 2\times2$; $P : 1 \rightarrow 16$). The computation needs only nine iterations with PE utilization rising to 100%. The tiling of POOM is directly applied to the innermost four loops (see Fig. 9), so the Trr $\times$ Tcc tiles would reuse the input tiles stored in the input register, without any extra access to the input buffer.

However, a too large P requires a higher bandwidth and larger capacity for the weight buffer. We should explore $\langle P$, Trr, Tcc $\rangle$ to find out the best POOM parameters that maximize PE utilization within the limited resources.

D. Support for FC Layers

FC layers can be represented using the same parameters as CONV layers, with additional constraints: $H = L = R = C = K = S = 1$. Since they have no kernel-level reuse, FC layers have an independent weight for each MAC operation, leading to a very large weight count. For example in AlexNet's FC layers, they require totally 58622k weights (25.1 times of CONV layers' weights), but their MAC operation count is only 8.8% of CONV layers. Therefore, FC layers produce large numbers of memory access, and their performance are usually bounded by the DRAM bandwidth [12].

Inspired by the kernel sharing of [13], we process FC layers in a batch to reuse weights and reduce the bandwidth requirement, thus improving the overall throughput. In one batch, the data sharing the same weight are regarded as one tile like in CONV layers. So the proposed hybrid data reuse and POOM can be applied to FC layers, by setting $R \times C =$ Tr $\times$ Tc $=$ batch size. The batch size ($= R \times C$) of FC layers can be set by the user or set as a scheduling parameter in the exploration space. In our experiment, it is explored using the scheduling framework to balance the overall throughput and batch processing latency, under the maximum DRAM bandwidth (12.8 GB/s for DDR3).

```
for(tm = m; tm < min(m + Tm); tm + +)          // Loop Tm
 for(tn = n; tn < min(n + Tn); tn + +)         // Loop Tn
  for(tr = r; tr < min(r + Tr, R); tr += trr)  // Loop Tr
   for(tc = c; tc < min(c + Tc, C); tc += tcc) // Loop Tc
    for(trr = tr; trr < min(tr + Trr, Tr); tr + +)  // Loop Trr
     for(tcc = tc; tcc < min(tc + Tcc, Tc); tc + +) // Loop Tcc
      {/*Convolution.*/}
```

Fig. 9. POOM: computation in the core.

IV. Architecture Design

In Section III, we find out that reconfigurable computation patterns are needed by the diversity of DCNNs and limited hardware resources. Thus, in this section, we design a reconfigurable architecture called DNA to support various computation patterns for different DCNNs, as shown in Fig. 10(a). Its main components are in accordance with the basic architecture in Fig. 3. DNA uses the configuration context stored in the controller to reconfigure its resources. The convolution core has two CEs. They share the Input REGs while each has independent Output REGs to store partial sums. A rectified linear unit and pooling unit are combined as the output unit for each CE.

A. Reconfigurable Data Path Design

As shown in Fig. 10(a), the total data buffer size is 256 kB. The input buffer and output buffer are integrated with unified addressing, so that their size can be adjusted for the diverse caching requirements of different data reuse patterns. To eliminate data loading latency, we apply "ping-pong" buffering techniques (in green) to the convolution core's Input/Output

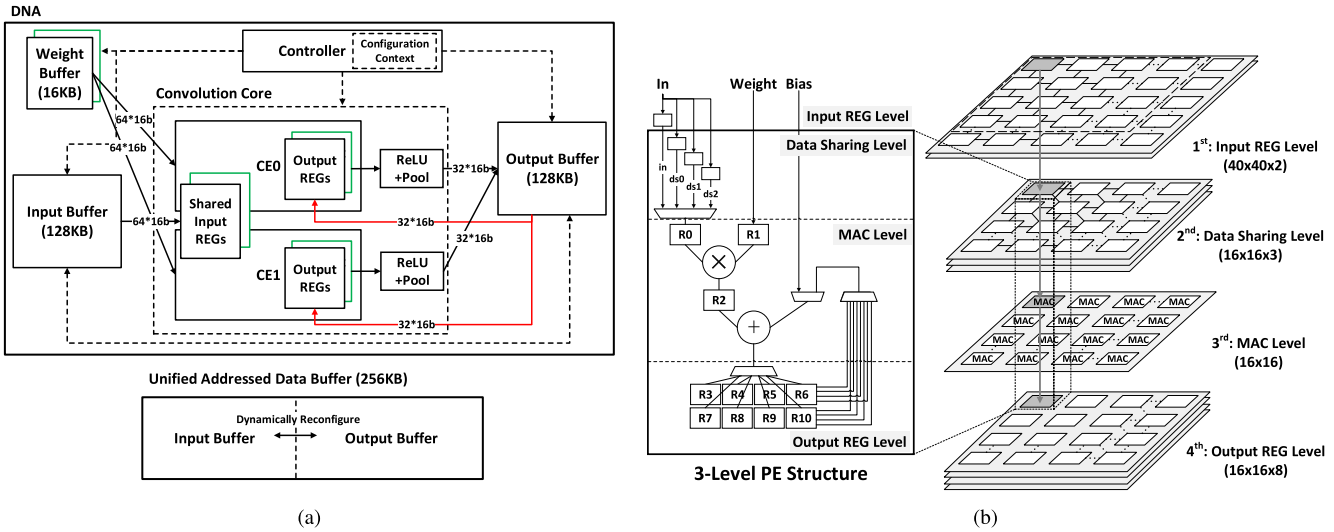


Fig. 10. DNA. (a) General architecture of DNA. (b) Four-level CE structure.

REGs and the weight buffer. The input/output buffer can also work in the "ping-pong" mode to exchange data with DRAM, if the caching requirement exceeds the buffer size. The data paths between memory and the convolution core can be easily configured to support the hybrid data reuse pattern (IR/OR/WR). In IR, the output buffer uses the feedback paths (in red) to write the previous partial sums back to each CE's Output REGs. In OR, the input buffer can write the next map to the "ping-pong" Input REGs, while the core is processing the current map. WR's data path is similar with IR's [see Fig. 5(e)], but the weights should be held in the buffer for full reuse. Slightly different from the basic architecture, the Output REGs are tightly coupled with CEs, so the partial sums can be quickly fetched for a new round of accumulations.

In DNA, the data buffer size (256 kB) is much larger than the weight buffer size (16 kB), because buffering data on-chip is more effective to save DRAM access. As demonstrated in Section III-B, DRAM access times depend on the computation pattern's data/weight buffering requirements. We witness that the buffering requirement for weights is usually very hard to meet, while the buffering requirement for data can be reached by adjusting scheduling parameters. For example, in the IR pattern, we need to store $M \cdot \mathrm{Tr} \cdot \mathrm{Tc}$ in the output buffer to minimize DRAM access for outputs ($\alpha_o = 0$). However, to reduce DRAM access for weights, we need to store all the weights (TW) in the buffer. TW is usually too large to be held on-chip (e.g., 600 kB for AlexNet's CONV2 layer), but we can select a proper Tr $\times$ Tc to make $M \cdot \mathrm{Tr} \cdot \mathrm{Tc}$ output data stored in the buffer. So, we allocate more space to data storage, and design a 16 kB weight buffer which is enough to meet the parallel access requirements of PEs.

B. Reconfigurable CE Design

1) Four-Level CE Structure: To support the proposed convolution mapping method (POOM), we design a four-level structure CE: input REG level, data sharing level, MAC level, and output REG layer, as shown in Fig. 10(b). The input REG level represents the shared Input REGs of the convolution core. The rest levels compose a 16×16 PE array of CE. Thus, each PE is a corresponding three-level structure, as shown on the left of Fig. 10(b).

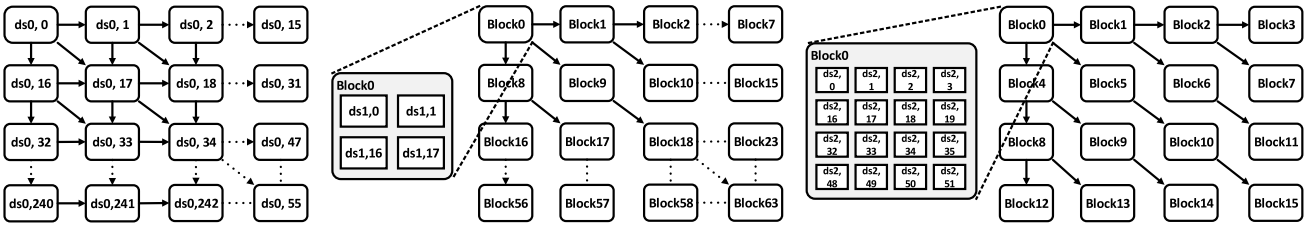
As demonstrated in Section III-C3, $\langle P, \mathrm{Trr}, \mathrm{Tcc} \rangle$ of POOM should be adjusted to maximize PE utilization under the hardware resource constraints. So, we design the data sharing level to enable CE's reconfigurability for different POOM parameters and fast data sharing in POOM, which will be further discussed in Section IV-B2.

The input REG level is made up of two 40×40 input register networks, serving as the shared input map for the two CEs. The input register network is a register array with 2-D-torus interconnections. So, the S-shaped sliding on input maps, required by OOM/POOM [see Fig. 7(a)], can be done by shifting register values horizontally and vertically. It is notable that the northwest 16×16 Input REGs can be directly accessed by the data sharing level and MAC level through Port "In" of each PE.

The MAC level is a cluster of 16×16 multiply accumulators (MAC), distributed in 16×16 PEs. Each MAC's input register R0 loads data from the input REG level (Port "in") and data sharing level (Port "ds0," "ds1," and "ds2"). Register R1 loads weights from the weight buffer (Port "Weight"). Each PE has eight Output REGs (Register $R3 \sim R10$) to store partial sums. So, all the 16×16 PEs' Output REGs form the output REG level with eight 16×16 registers.

2) Data Sharing Networks: As demonstrated in Section III-C3, different blocks in POOM need to share the same input data to improve PE utilization. Meanwhile, $\langle P, \mathrm{Trr}, \mathrm{Tcc} \rangle$ should be adjusted under the hardware resource constraints. Thus, in the data sharing level, we designed three data sharing networks (DSNs) to enable flexible and fast data transferring between different blocks (see Fig. 11).

DSN0 is a mesh-like register array of strengthened with diagonal interconnections, as shown in Fig. 11(a). It perfectly matches the data sharing pattern in large stride cases, because

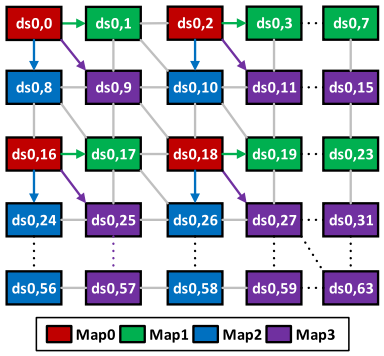


(a) (b) (c)

Fig. 11. DSNs: DSN0 is a 16×16 sharing network with 1×1 blocks; DSN1 is an 8×8 sharing network with 2×2 blocks (Trr = Tcc = 2); DSN2 is a 4×4 sharing network with 4×4 blocks (Trr = Tcc = 4). (a) DSN0: 16×16 network. (b) DSN1: 8×8 network. (c) DSN2: 4×4 network.

each register value can be directly transferred to its three neighbors in the east, south, and southeast. We demonstrate how DSN0 works with a large stride case [stride ($S = 2$)]. DSN0 is configured as the sharing pattern in Fig. 12(a). In each cycle, the "red" registers collect input data from the input REG level, and then transfer them to their nearest three neighbors. Meanwhile, each MAC loads its input data from Port "ds0," and multiplies it by its corresponding kernel weights. Fig. 12(b) shows the detailed data flows of PE0, PE1, PE8, and PE9, from four different output maps. The whole process is fully pipelined, so the PE utilization reaches almost 100%, with the help of POOM and DSN0. Moreover, since each register can share its value with the three neighbors in the east, south, and southeast, DSN0 can be reconfigured to support large strides from 2 to 16, covering all the state-of-the-art DCNNs.

To support tiling irregular output maps with Trr $\times$ Tcc $=$ 2×2 and 4×4, we design DSN1 and DSN2 to enable their data sharing patterns. DSN1 is a 8×8 sharing network with 2×2 blocks. In each cycle, Block0 loads 2×2 data from the input REG level, and transfers them to the other blocks with the interconnections of DSNs. Each block feeds the data to its corresponding 2×2 MACs through Port "ds1." In this way, 8×8 PE blocks always share the same input data, and compute at most 64 different output maps in parallel with Trr $\times$ Tcc $=$ 2×2. Similarly, DSN2 is a 4×4 sharing network with 4×4 blocks, enabling data sharing in Trr $\times$ Tcc $= 4 \times 4$. We did not design a 2×2 sharing network with 8×8 blocks, because we find out that tiling with Trr $\times$ Tcc $= 8 \times 8$ cannot improve PE utilization. Therefore, with the above-mentioned three DSNs, CE can be reconfigured for different POOM parameters of Trr $\times$ Tcc $= 1 \times 1, 2 \times 2$, and 4×4.



(a)

Cycle		1	2	3	4	5
PE0	In	I0	I1	I9	I8	...
	Weight	W_0^0	W_1^0	W_3^0	W_2^0	...
	ds0	I0	I1	I9	I8	...
PE1	In	I1	I2	I10	I9	...
	Weight		W_0^1	W_1^1	W_3^1	W_2^1
	ds0		I0	I1	I9	I8
PE8	In	I8	I9	I17	I16	...
	Weight		W_0^2	W_1^2	W_3^2	W_2^2
	ds0		I0	I1	I9	I8
PE9	In	I9	I10	I18	I17	...
	Weight		W_0^3	W_1^3	W_3^3	W_2^3
	ds0		I0	I1	I9	I8

(b)

Fig. 12. Case study on POOM+DSN0: $S = 2$, $K = 2$, $A = 8$, Tr = Tc = Trr = Tcc = 4, and $P = 4$. (a) Data sharing pattern on DSN0 to support POOM. (b) Data flows of PE0, PE1, PE8, and PE9, respectively, for four output maps. W_j^i refers to the ith weight of Map j.

C. DNA's Work Flow and Scheduling Framework

DNA's two-phase work flow is concluded in Fig. 13. Phase1 is compiling a DCNN into configurations for DNA, while Phase2 is executing the DCNN on the DNA system.

As we have mentioned earlier, the diversity of DCNN's layer sizes and convolution parameters demand flexible and efficient computation patterns to optimize both power efficiency and performance. We propose a layer-based scheduling framework for Phase1, which assigns the best computation pattern to each layer of a given DCNN. For each layer, the scheduling is formulated as a dual-objective optimization problem, as shown in Fig. 13. R_i, R_o, and B_w are the capacity of the Input REGs, Output REGs, and weight buffer, respectively, measured in the data count. Meanwhile, the DRAM bandwidth requirement BW_{req} should not exceed the available maximum DRAM bandwidth BW_{max}. With this framework, we can obtain the best computation pattern for each layer with the optimal power efficiency and performance, under DNA's hardware constraints. The computation pattern is described in a data reuse pattern ⟨IR/OR/WR, Tr, Tc, Tm, Tn⟩ combined with a convolution mapping method ⟨P, Trr, Tcc⟩. Notice that we usually transform the dual-objective optimization into a single-objective optimization problem. The single objective can be changed for different design considerations. For example, if we want to balance power efficiency and performance, we can set

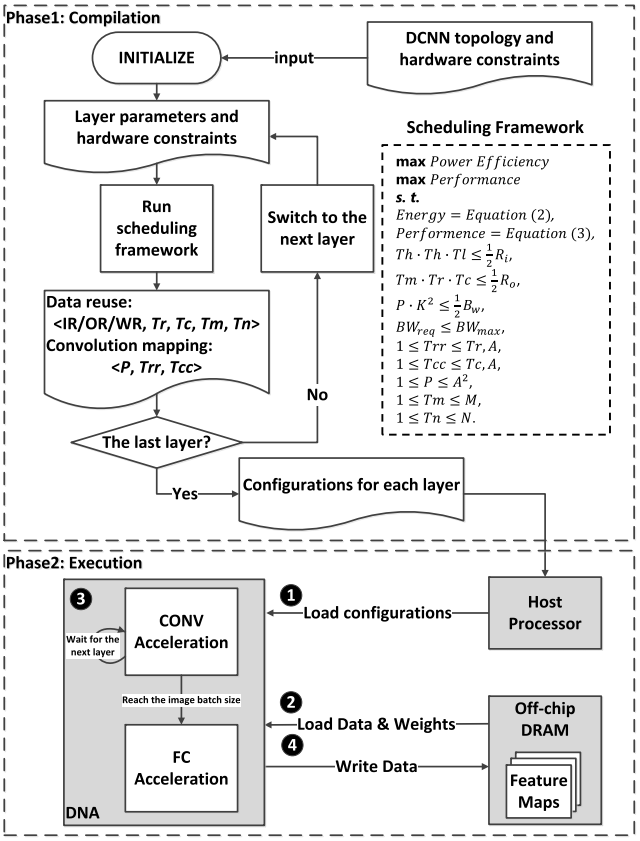


Fig. 13. DNA's work flow.

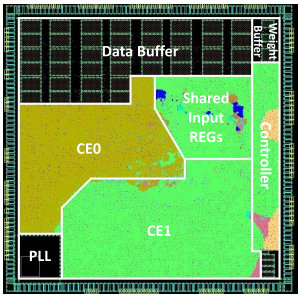
the single objective as the objectives' product or energy delay product; if we pay more attention to performance, we can optimize performance first and then optimize power efficiency. In the end of Phase1, the scheduling result is compiled into the configuration context.

Phase2 has four stages: 1) DNA loads the previously compiled configurations from the host processor; 2) DNA loads data and weights from the off-chip DRAM; 3) DNA executes the current layer (CONV or FC); and 4) DNA writes the layer's outputs into DRAM. The four stages are fully pipelined and run until a batch of feature maps are all processed. As illustrated in stage 3), DNA accelerates each CONV layer sequentially, and then process FC layers in a batch.

V. Experimental Results

A. Experiment Setup

We implement DNA using the Synopsys Design Compiler with the TSMC 65-nm LP technology. Functional simulations and performance measurements are conducted on Synopsys VCS, and DNA's chip-only power consumption is estimated by PrimeTime PX. We select four famous DCNN models as the benchmarks: AlexNet [6], VGG19 [14], GoogLeNet [15], and ResNet50 [16]. We schedule DNA with the proposed framework for each benchmark, targeting balance between power efficiency and performance. DNA is then reconfigured to run the above-mentioned benchmarks. As for the batch size for FC layers, it is explored using the scheduling framework to balance the overall throughput and batch processing latency,



Technology	TSMC 65nm LP
Supply Voltage	IO: 3.3V Core: 1.2V
Area	4.0mm x 4.0mm
SRAM	280KB
Frequency	200MHz
Peak Performance	204.8GOPS
Average* Performance	194.4GOPS
Average* Power	479mW
Precision	16-bit Fixed Point

*: Average results measured on the benchmarks.

Fig. 14. DNA's layout and characteristics.

under the maximum DDR3 bandwidth (12.8 GB/s). The batch sizes for AlexNet, VGG19, GoogLeNet, and ResNet50, are 16, 16, 64, and 64 respectively. In the scheduling framework, we estimate each data reuse pattern's memory access energy with (11). The DRAM access is set as 70 pJ/bit (typical access energy of DDR3 [10]).

The experimental methodology is as follows. We analyze DNA's characteristics after layout in Section V-B. Memory access and total energy consumption are compared with the conventional methods (IR/OR/WR+OOM) in Section V-C. Performance evaluation is then conducted in Section V-D. In Sections V-C and V-D, we first provide a general analysis on all the benchmarks, and then perform a breakdown analysis on each layer of AlexNet. Finally, in Section V-E, we compare DNA with the state-of-the-art designs on general-purpose acceleration platforms (CPU and GPU), and reconfigurable architectures (FPGA and ASIC).

B. Characteristics After Layout

Fig. 14 shows DNA's layout and basic characteristics in the TSMC 65-nm LP technology. The chip area is 4×4 mm^2 with 280-kB SRAM. The peak performance is 204.8 GOPS at 200 MHz, when the PE utilization is 100%. The average performance and power consumption are 194.4 GOPS and 479 mW, measured on DNA's netlist combined with the four benchmarks' simulation wave files. The data precision is 16-bit fixed point, which guarantees no quality loss in hardware implementation.

C. Memory Access and Energy Consumption Analysis

1) General Analysis on the Benchmarks: Fig. 15 shows our method's memory access reduction and energy savings over conventional methods (IR/OR/WR+OOM) on the benchmarks. In conventional IR/OR/WR, they maximize $\mathrm{Th} \times \mathrm{Tl}$ or $\mathrm{Tr} \times \mathrm{Tc}$ to perform convolutions on larger feature maps and maximize intramap data reuse. In our hybrid reuse pattern, we explore the above-mentioned tiling parameters and measure each pattern's memory access energy with (11). Conventional IR, OR, and WR are only three single points in our exploration space, but DNA is configured with the best data reuse pattern for each layer. Our method achieves mean buffer access reduction of 1.4 times over IR, 1.7 times over OR, and 2.2 times over WR [see Fig. 15(a)]. As for DRAM access, its reduction

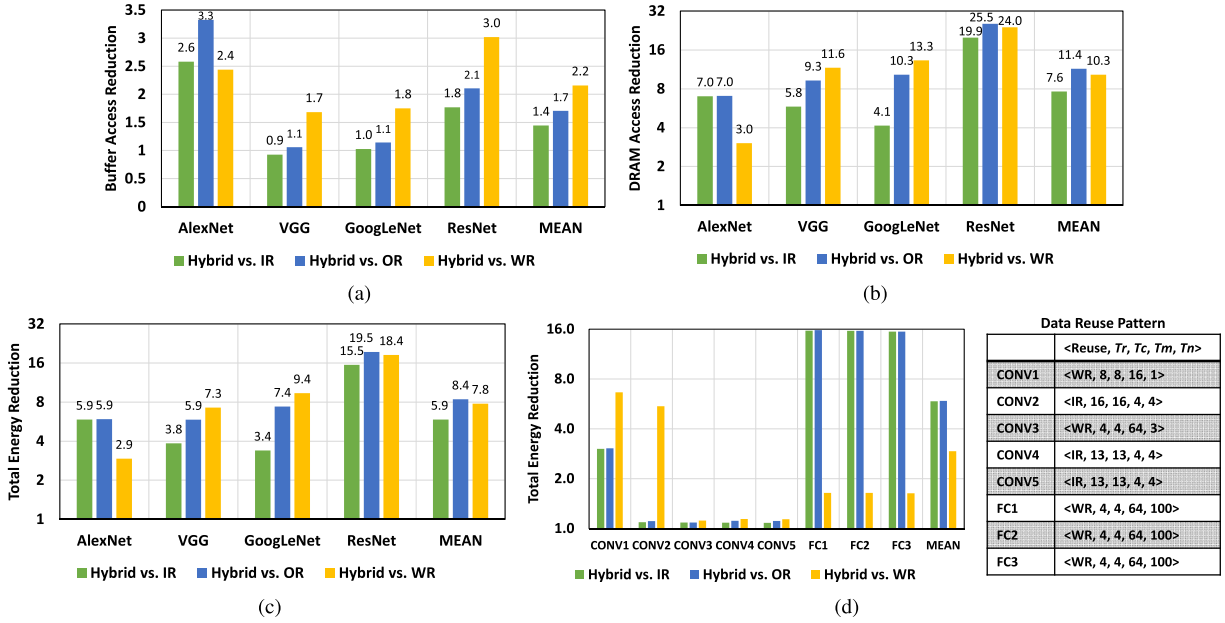


Fig. 15. Memory access and total energy reduction: hybrid data reuse pattern over conventional IR/OR/WR. (a) Buffer access reduction. (b) DRAM access reduction. (c) Total energy reduction. (d) Total energy reduction: breakdown analysis on AlexNet.

is 7.6 times over IR, 11.4 times over OR, and 10.3 times over WR [see Fig. 15(b)].

Besides memory access comparison, we also present total energy comparison in a batch, as shown in Fig. 15(c). In the experiment, we do not use (14) to estimate energy. Instead, total energy consumption is measured by the following equation, based on the simulation results:

$$\text{Energy} = \text{Power}_{\text{DNA}} \cdot \text{Time} + \text{MA}_{\text{DRAM}} \cdot E_{\text{DRAM}}. \quad (14)$$

$\text{Power}_{\text{DNA}}$ and time are the power of DNA and total execution time when running the benchmarks. They are measured on the DNA's simulation wave files, with PrimeTime PX and Synopsys VCS. MA_{DRAM} is estimated with the DRAM access model proposed in Section III-B. E_{DRAM} is estimated as 70 pJ/bit, typical access energy of DDR3 [17]. We witness that energy reduction is almost proportional to DRAM access reduction, averagely 5.9 times over IR, 8.4 times over OR, and 7.8 times over WR [see Fig. 15(c)]. It is because DRAM access power takes up a large part of the total power consumption. On the four benchmarks, DRAM access power is 793 mW on average, while the mean system-level power consumption is 1272 mW.

2) Breakdown Analysis on AlexNet: A layerwise breakdown analysis on total energy consumption is given in Fig. 15(d). For the five CONV layers and three FC layers of AlexNet, our data reuse pattern is WR→IR→WR→IR→IR→WR→WR→WR, with different Tr × Tc for each layer. On CONV1 and CONV2, our method shows more significant energy reduction, because conventional methods have higher data buffering requirements that generate more DRAM access. FC layers are memory-intensive due to the large numbers of weights. WR can reuse weights in a batch to reduce repeated DRAM access for weights, but IR/OR cannot. So our method selects WR as the data reuse pattern for FC layers.

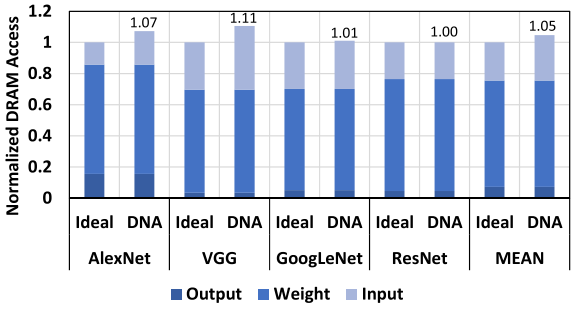


Fig. 16. Input tile overlapping's influence on DRAM access.

3) Influence of Input Tile Overlapping: As mentioned in Section III-B, the input tiles' overlapping regions increase TI by 1∼1.5 times. Since the total energy consumption is mainly determined by DRAM access, we study the tile overlapping's influence on DRAM access, as shown in Fig. 16. We compare DNA's DRAM access with that of the ideal case, in which the computation patterns are the same but TI is not enlarged by tile overlapping. On the four benchmarks, DNA's DRAM access times are slightly higher than the ideal case, with an average increase of only 5% (see Fig. 16). The reason is that DRAM access for input data takes up only a small proportion (24.5% in the ideal case) of the total access, while the rest DRAM access is for weights and outputs. To reach the ideal case, each layer's input data should be totally stored in the input buffer for reuse. However, as modern DCNN models become more complex, many of their layers' inputs are over 500 kB or even 1 MB, much more than the buffer capacity of a small-footprint chip like DNA. However, with the proposed method, memory access times are reduced and the tile overlapping's influence is alleviated even with limited buffer capacity.

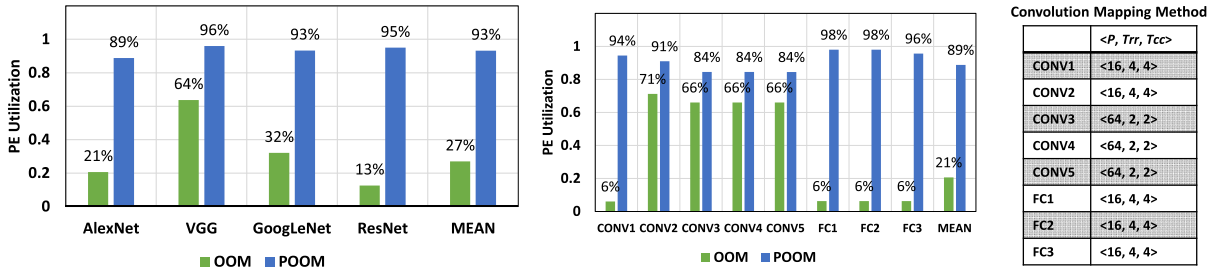


Fig. 17. PE utilization comparison: OOM versus POOM. (a) General analysis on the benchmarks. (b) Breakdown analysis on AlexNet.

TABLE II
COMPARISON WITH STATE-OF-THE-ART DESIGNS ON ALEXNET

	Xeon E5-2620		NVIDIA K40 [4]		FPGA'15 [5]		ISSCC'16 [8]		DNA	
AlexNet Layers	Time (ms)	Speedup	Time (ms)	Speedup	Time (ms)	Speedup	Time (ms)	Speedup	Time (ms)	Speedup
CONV1	14.78	1x	0.26	56.8x	7.67	1.9x	5.23	2.8x	1.09	13.6x
CONV2	27.35	1x	0.53	51.6x	5.35	5.1x	10.48	2.6x	2.40	11.4x
CONV3	21.84	1x	0.95	23.0x	3.79	5.8x	5.90	3.7x	1.73	12.6x
CONV4	16.69	1x	0.87	19.2x	2.88	5.8x	4.60	3.6x	1.30	12.8x
CONV5	11.64	1x	0.66	17.6x	1.93	6.0x	2.63	4.4x	0.86	13.5x
CONV Total	92.30	1x	3.27	28.2x	21.62	4.3x	28.84	3.2x	7.38ms	12.5x
PE Utilization	-		-		68.8%		68.7%		88.1%	
Performance (GOPS)	14.4		407.3		61.62		46.2		180.4	
Power[a] (W)	s: 80		s: 235		s: 18.61		s: 0.577, c: 0.278		s: 1.226, c: 0.434	
Efficiency[a] (GOPS/W)	s: 0.18		s: 1.7		s: 3.3		s: 80.0, c: 166.2		s: 147.2, c: 415.9	
FC1	8.74	1x	0.14	60.4x	-	-	-	-	0.38	23.2x
FC2	4.02	1x	0.07	59.8x	-	-	-	-	0.17	24.1x
FC3	1.04	1x	0.02	45.5x	-	-	-	-	0.04	24.8x
FC Total	13.80	1x	0.23	58.8x	-	-	-	-	0.59	23.6x
Total	106.09	1x	3.49	30.4x	-	-	-	-	7.91	13.3x
Performance (GOPS)	13.7		414.6		-		-		181.8	
Power[a] (W)	s: 80		s: 235		-		-		s: 1.665, c: 0.436	
Efficiency[a] (GOPS/W)	s: 0.17		s: 1.8		-		-		s: 109.2, c: 416.6	

[a] Abbreviation for power measurement types: s (system-level power considering off-chip memory), c (chip only power).

D. Performance Evaluation

1) General Analysis on the Benchmarks: Besides memory access comparison, we evaluate the performance improvements of the proposed method. Fig. 17(a) shows the PE utilization of conventional OOM and the proposed POOM. In OOM, PE utilization on AlexNet, GoogLeNet, and ResNet is relatively lower than that on VGG, because these three networks have several layers with stride $S > 1$: AlexNet has a layer with $S = 4$; VGG has a layer with $S = 2$; ResNet has seven layers with $S = 2$. According to the analysis in Section III-C2, such large stride layers would result in PE utilization of nearly $(1/S^2)$. Moreover, the mismatch between irregular feature map size (Tr $\times$ Tc) and the fixed array size ($A \times A$) further limits OOM's PE utilization (39% on average). With a more scalable convolution mapping method POOM, we solve the above-mentioned problems and achieve mean PE utilization of 93%.

2) Breakdown Analysis on AlexNet: We also provide a breakdown analysis on AlexNet, as shown in Fig. 17(b). The POOM scheduling result on each layer is presented on the right. OOM has quite low PE utilization in CONV1 because of its large stride $S = 4$. Through POOM, we compute 16 maps in parallel ($P = 16$) to make better use of the PEs, thus achieving 94% utilization on CONV1. In CONV2, we use POOM to solve the mismatch problem between the 27×27 map size and 16×16 array size, obtaining 91% PE utilization. For CONV3∼5, DNA is reconfigured with $\langle P$, Trr, Tcc $\rangle = \langle 64, 2, 2 \rangle$ to obtain higher utilization than OOM. POOM is also useful for FC layers. The batch size for AlexNet is 16 in our scheduling, Thus in one batch, the data sharing the same weight are regarded as one 4×4 tile like in CONV layers. In conventional OOM, only one tile is processed each time (PE utilization = 6%). By setting $P = 16$ in POOM, at most 16 tiles can be processed in parallel for higher PE utilization (98%). DNA achieves average PE utilization of 89% on AlexNet, which is 4.2 times over conventional OOM.

E. Design Comparison

We compare DNA with four state-of-the-art implementations on CPU, GPU, FPGA, and ASIC with the same benchmark AlexNet, as shown in Table II. The baseline is a Caffe [21] version of AlexNet, running on CPU (Intel Xeon E5-2620 v2, clocked at 2.10 GHz). The GPU platform is NVIDIA K40 GPU with 12-GB memory. Reference [5] is an implementation on Xilinx VC707, clocked at 100 MHz. Reference [8] is an ASIC design for DCNN acceleration,

TABLE III
GENERAL COMPARISON WITH STATE-OF-THE-ART DESIGNS

	CPU	GPU	FPGA'15 [5]	ICCAD'16 [18]		ISSCC'16 [8]	ISCA'16 [19]		DNA
Platform	Xeon E5-2620	NVIDIA K40	Virtex VX485t	Ultrascale KU060	Virtex VX690t	ASIC 65nm	ASIC 28nm	ASIC 15nm	ASIC 65nm
Benchmark	AlexNet, VGG19, GoogLeNet, ResNet50 [6, 14–16]		AlexNet (CONV) [6]	VGG16 [14]		AlexNet (CONV) [6]	Scene Labeling [20]		AlexNet, VGG19, GoogLeNet, ResNet50 [6, 14–16]
Performance[a] (GOPS)	16.0	390.0	61.62	266	354	46.2	8.0	132.4	194.4
Power[ab] (W)	s: 80	s: 235	s: 18.61	s: 25	s: 26	c: 0.278 s: 0.577	c: 0.25 s: 1.86	c: 3.41 s: 21.50	c: 0.479 s: 1.272
Efficiency[ab] (GOPS/W)	s: 0.2	s: 1.7	s: 3.3	s: 10.6	s: 13.6	c: 166.2 s: 80.0	c: 31.9 s: 4.3	c: 38.8 s: 6.2	c: 406.2 s: 152.9

[a] Average results, measured on the benchmarks.
[b] Abbreviation for power measurement types: s (system-level power considering off-chip memory), c (chip only power).

working at 200 MHz. The top half of Table II presents the results on CONV layers, and the bottom half of Table II presents the results on FC layers and the whole network. Notice that [5] and [8] only have results on CONV layers in their papers. We measure their performance with the operations per second during the total execution time. Each multiply accumulation is regarded as two operations in our evaluation. Besides chip-only power, we also present system-level power that considers off-chip DRAM in Table II. The corresponding power efficiency is measured in GOPS/W.

On AlexNet's CONV layers, DNA's PE utilization is 88.1%, which is almost 1.3 times of [5] and [8], owing to POOM. Chen *et al.*'s [8] DRAM access on the CONV layers is reported to be 15.4 MB, while DNA's DRAM access is only 10.4 MB. Therefore, DNA achieves 147.2 GOPS/W on the CONV layers, which is 1.8 times of [8] and 44.6 times of [5]. On the whole AlexNet, DNA achieves speedup of 13.3 times over the baseline. The GPU design achieves the highest performance of all, but its power consumption reaches 235 W, leading to power efficiency of only 1.8 GOPS/W. In comparison, DNA consumes only 436-mW on-chip and 1.665 W if considering DRAM access power, due to the dedicated PE and data path design. DNA's system-level power efficiency for the whole AlexNet is 109.2 GOPS/W, which is 642.4 times of the baseline and 60.7 times of GPU.

We conduct a more general analysis with the state-of-the-art designs, on more benchmarks, as shown in Table III. A recent FPGA-based design [18] and a recent ASIC design [19] are included in the comparison. We compare the works' average performance, power, and power efficiency, measured on their benchmarks (listed in Row "Benchmark" of TABLE III). Compared with [18], DNA's system-level power efficiency is 14.4 times over the KU060 version, and 11.2 times over the VX690t version. Reference [19] is a design integrated with 160 GB/s high-density 3-D memory. DNA's performance is 24.3 times over the 28-nm version and 1.5 times over the 15-nm version, with, respectively, 35.6 times and 24.7 times higher system-level power efficiency. Although DNA has a lower DRAM bandwidth (12.8 versus 160 GB/s), we can reduce DRAM access and DRAM bandwidth requirements, with the proposed hybrid data reuse.

VI. CONCLUSION

In this paper, we implement a reconfigurable DCNN architecture named DNA with the TSMC 65-nm technology. It can be reconfigured with optimized computation patterns for different DCNN models. The proposed method achieves total energy reduction of 5.9 times over IR, 8.4 times over OR and 7.8 times over WR, with mean PE utilization of 93% on the benchmarks. On the benchmarks, DNA achieves average performance of 194.4 GOPS at 200 MHz and consumes only 479 mW. Its system-level power efficiency is 152.9 GOPS/W, which is one to two orders higher than the state-of-the-art designs.

REFERENCES

[1] Q. V. Le, "Building high-level features using large scale unsupervised learning," in *Proc. IEEE Int. Conf. Acoust., Speech Signal Process. (ICASSP)*, May 2013, pp. 8595–8598.
[2] A. Coates, B. Huval, T. Wang, D. Wu, B. Catanzaro, and N. Andrew, "Deep learning with COTS HPC systems," in *Proc. Int. Conf. Mach. Learn. (ICML)*, 2013, pp. 1–9.
[3] J. Cong and B. Xiao, "Minimizing computation in convolutional neural networks," in *Proc. Int. Conf. Artif. Neural Netw. (ICANN)*, 2014, pp. 281–290.
[4] "Tesla K40 GPU active accelerator," NVIDIA, Santa Clara, CA, USA, Tech. Rep. BD-06949-001_v03, 2013.
[5] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays (FPGA)*, 2015, pp. 161–170.
[6] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Proc. Adv. Neural Inf. Process. Syst. (NIPS)*, 2012, pp. 1097–1105.
[7] T. Chen *et al.*, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *Proc. ACM Int. Conf. Archit. Support Program. Lang. Oper. Syst. (ASPLOS)*, 2014, pp. 269–284.
[8] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," in *IEEE Int. Solid-State Circuits Conf. (ISSCC) Dig. Tech. Papers*, Jan. 2016, pp. 262–263.
[9] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," in *Proc. Int. Conf. Comput. Vis. (ICCV)*, 2015, pp. 1026–1034.
[10] K. T. Malladi, B. C. Lee, F. A. Nothaft, C. Kozyrakis, K. Periyathambi, and M. Horowitz, "Towards energy-proportional datacenter memory with mobile dram," in *Proc. Int. Symp. Comput. Archit. (ISCA)*, 2012, pp. 37–48.
[11] A. Rahman, J. Lee, and K. Choi, "Efficient FPGA acceleration of convolutional neural networks using logical-3D compute array," in *Proc. IEEE Design, Autom. Test Eur. Conf. Exhibit. (DATE)*, Mar. 2016, pp. 1393–1398.
[12] J. Qiu *et al.*, "Going deeper with embedded fpga platform for convolutional neural network," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays (FPGA)*, 2016, pp. 26–35.

[13] L. Song, Y. Wang, Y. Han, X. Zhao, B. Liu, and X. Li, "C-brain: A deep learning accelerator that tames the diversity of cnns through adaptive data-level parallelization," in *Proc. Design Autom. Conf. (DAC)*, 2016, pp. 123:1–123:6.
[14] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *Proc. Int. Conf. Learn. Represent. (ICLR)*, 2014.
[15] C. Szegedy *et al.*, "Going deeper with convolutions," in *Proc. Comput. Vis. Pattern Recognit. (CVPR)*, 2015, pp. 1–9.
[16] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. Comput. Vis. Pattern Recognit. (CVPR)*, 2016, pp. 770–778.
[17] *DDR3 SDRAM, JESD79-3F*, accessed on Jul. 2012. [Online]. Available: http://www.jedec.org/standards-documents/docs/jesd-79-3d
[18] C. Zhang, Z. Fang, P. Zhou, P. Pan, and J. Cong, "Caffeine: towards uniformed representation and acceleration for deep convolutional neural networks," in *Proc. Int. Conf. Comput.-Aided Design (ICCAD)*, 2016, pp. 1–8.
[19] D. Kim, J. Kung, S. Chai, S. Yalamanchili, and S. Mukhopadhyay, "Neurocube: A programmable digital neuromorphic architecture with high-density 3D memory," in *Proc. IEEE Int. Symp. Comput. Archit. (ISCA)*, Jun. 2016, pp. 380–392.
[20] S. Gould, R. Fulton, and D. Koller, "Decomposing a scene into geometric and semantically consistent regions," in *Proc. Int. Conf. Comput. Vis. (ICCV)*, 2009, pp. 1–8.
[21] Y. Jia *et al.*, "Caffe: Convolutional architecture for fast feature embedding," in *Proc. ACM Int. Conf. Multimedia (MM)*, 2014, pp. 675–678.

Fengbin Tu received the B.S. degree in electronic science and technology from the Beijing University of Posts and Telecommunications, Beijing, China, in 2013. He is currently pursuing the Ph.D. degree with the Institute of Microelectronics, Tsinghua University, Beijing.

His current research interests include computer architecture, deep learning, VLSI design, and approximate computing.

Shouyi Yin (M'09) received the B.S., M.S., and Ph.D. degrees in electronic engineering from Tsinghua University, Beijing, China, in 2000, 2002, and 2005, respectively.

He was with the Imperial College London, London, U.K., as a Research Associate. He is currently with the Institute of Microelectronics, Tsinghua University, as an Associate Professor. His current research interests include reconfigurable computing, mobile computing, and SoC design.

Peng Ouyang received the B.S. degree in electronic and information technology from Central South University, Changsha, China, in 2008, and the Ph.D. degree in electronic science and technology from Tsinghua University, Beijing, China, in 2014.

He currently holds a post-doctoral position with the School of Information, Tsinghua University. His current research interests include the computer vision, machine learning, vision processor, and reconfigurable computing.

Shibin Tang received the Ph.D. degree in computer architecture from the Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China.

He is currently a Post-Doctoral Researcher with the Institute of Microelectronics, Tsinghua University, Beijing. His current research interests include deep learning accelerator architecture, computer architecture, cache coherence, and memory consistency.

Leibo Liu (M'10) received the B.S. degree in electronic engineering from Tsinghua University, Beijing, China, in 1999, and the Ph.D. degree from the Institute of Microelectronics, Tsinghua University, in 2004.

He is currently an Associate Professor with the Institute of Microelectronics, Tsinghua University. His current research interests include reconfigurable computing, mobile computing, and VLSI design.

Shaojun Wei (M'91) received the Ph.D. degree from the Faculté polytechnique de Mons, Mons, Belgium, in 1991.

He became a Professor with the Institute of Microelectronics, Tsinghua University, Beijing, China, in 1995. His current research interests include VLSI SoC design, EDA methodology, and communication ASIC design.

Dr. Wei is a Senior Member of the Chinese Institute of Electronics.