CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2017; **29**:e3850
Published online 6 May 2016 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cpe.3850

SPECIAL ISSUE PAPER

FPGA-accelerated deep convolutional neural networks for high throughput and energy efficiency‡

Yuran Qiao*,†, Junzhong Shen, Tao Xiao, Qianming Yang, Mei Wen and Chunyuan Zhang

Department of Computer, State Key Laboratory of High Performance Computing, National University of Defense Technology, Changsha, Hunan, China

SUMMARY

Recent breakthroughs in the deep convolutional neural networks (CNNs) have led to great improvements in the accuracy of both vision and auditory systems. Characterized by their deep structures and large numbers of parameters, deep CNNs challenge the computational performance of today. Hardware specialization in the form of field-programmable gate array offers a promising path towards major leaps in computational performance while achieving high-energy efficiency.

In this paper, we focus on accelerating deep CNNs using the Xilinx Zynq-zq7045 FPGA SoC. As most of the computational workload can be converted to matrix multiplications, we adopt a matrix multiplier-based accelerator architecture. Dedicated units are designed to eliminate the conversion overhead. We also design a customized memory system according to the memory access pattern of CNNs. To make the accelerator easily usable by application developers, our accelerator supports Caffe, which is a widely used software framework of deep CNN. Different CNN models can be adopted by our accelerator, with good performance portability. The experimental results show that for a typical application of CNN, image classification, an average throughout of 77.8 GFLOPS is achieved, while the energy efficiency is 4.7× better than an Nvidia K20 GPGPU. © 2016 The Authors. *Concurrency and Computation: Practice and Experience* Published by John Wiley & Sons Ltd.

Received 27 August 2015; Revised 3 April 2016; Accepted 3 April 2016

KEY WORDS: CNN; Accelerator; FPGA; Matrix Multiplier; Caffe

1. INTRODUCTION

Emerging applications such as micro-UAV (unmanned aerial vehicle), domestic robot and internet media data analysis need fast computing systems to perceive the real world. Deep neural networks achieve state-of-the-art perception in both vision and auditory systems [1, 2]. Meanwhile, characterized by their deep structures and large numbers of parameters, deep neural networks challenge the today's computational performance. Take the well-known model, AlexNet proposed in [1] as an example, 1.36 $\times 10^9$ operations (usually 32-bit single-precision floating-point operations) with 60 million parameters are needed by one forward propagation alone. A typical vision perception application, object detection, using the model repeatedly usually requires a computation capability of 10^9 to 10^{12} operations per second [3].

Recent works have revealed that the application domain of deep neural networks is crossing from embedded systems to data centers [4, 5]. Both of them have stringent energy constraint, which

*Correspondence to: Yuran Qiao, Department of Computer, State Key Laboratory of High Performance Computing, National University of Defense Technology, Changsha, Hunan, China.

†E-mail: qiaoyuran@nudt.edu.cn

‡The copyright line for this article was changed on 16 June 2016, after original online publication.

general-purpose accelerators[6, 7] (such as GPGPU [8]) can hardly satisfy. Specific accelerator is thus becoming a research hot spot.

Specialized accelerator in the form of Filed Programmable Gate Array (FPGA) offers a promising path towards major leaps in performance and energy efficiency. In this paper, we focus on accelerating the forward propagation of deep Convolutional Neural Networks (CNNs) using an FPGA-based accelerator.

We adopt a matrix multiplier-based accelerator architecture which has two superiorities: flexibility and consistency. The structures of different CNNs may differ greatly. CNN accelerators for fixed network structures [4, 9] are inflexible. A common design that can efficiently handle variable network structures is needed. Recently, the authors of [10] discussed the design space of a CNN accelerator, but it still needs to adjust the accelerator structure to a fixed network. Converting convolutions to matrix multiplications (we will use 'unrolling' to describe it) is a good choice to handle broader spectrum of network structures [11, 12]. With respect to consistency, traditional works [4, 9, 10] design different accelerating units for different parts of CNNs. As most of the computational workload of a CNN exists in convolutional layers and full connection layers, the matrix multiplier-based accelerator can handle both the two kinds of layers. Compared with traditional architectures, the on-chip resources used to build accelerating module for full connection layers are saved.

It is a common practice to use some developed software framework running on CPUs or GPUs (such as Caffe [13], Torch [14]) to develop CNN applications at first. Most of the existing CNN accelerators use custom frameworks for simplicity, which makes it difficult to take advantage of the rich achievements based on widely used frameworks. We assert that it is important to make the accelerator supports some representative developed frameworks. In this paper, we choose Caffe, a popular high-performance CNN framework implementation.

In the process of designing our accelerator, we see that there are some challenges: (1) Overhead of unrolling the convolutions to matrix multiplications can not be ignored. If we only use a matrix multiplier to accelerate the matrix multiplication part and still leave the unrolling part to the host processor, the upper bound of speed-up is limited according to the Amdahl's law. (2) The non-sequential external memory access brings an obvious memory bandwidth slow-down that greatly limits the accelerator's performance. (3) The unrolled matrix multiplications in different CNN structures have different sizes. It is challenging to efficiently map these matrix multiplications of different sizes to a fixed structure accelerator. (4) The Caffe framework relies on Linux, which has a memory management mechanism. FPGA accelerator accesses Dynamic Random Access Memory (DRAM) through physical space, but the Caffe framework accesses DRAM through the operating system's user space. The data communication between the different spaces may cause extra overhead.

To overcome these challenges, some novel contributions are made by this paper:

- A stream-mapper unit is designed to handle the convolution unrolling task. For the reason that convolution unrolling and matrix multiplication can be overlapped simultaneously, the time overhead of unrolling is eliminated.
- We propose a prefetch strategy and implement a prefetch unit structure to make the address stream to the external memory sequential. As memory access with sequential addresses can benefit from the burst characteristic of the memory port, the memory bandwidth can be fully used.
- We optimize the blocking strategy to make matrix multiplications of different sizes perform efficiently.
- We propose a novel memory management scheme for our FPGA accelerator to efficiently share the data with the host processor.

We implement an FPGA-extended version Caffe based on two Xilinx Zynq-zq7045 FPGA SoC chips using 1600 Dsp48Es (the basic computational resource of the Xilinx FPGA) at 150 MHz clock rate. A performance of 77.8 Gflops is achieved. Compared with an Intel Xeon X5675 (3.4 Ghz, 6-core) processor, 3.54× speed-up is achieved. The energy efficiency is also better than an Nvidia K20 GPGPU by a factor of 4.7×.

The rest of this paper is organized as follows: In Section 2, brief background knowledge is presented. Section 3 describes our design and implementation details. Section 4 shows the experiment

DOI: 10.1002/cpe

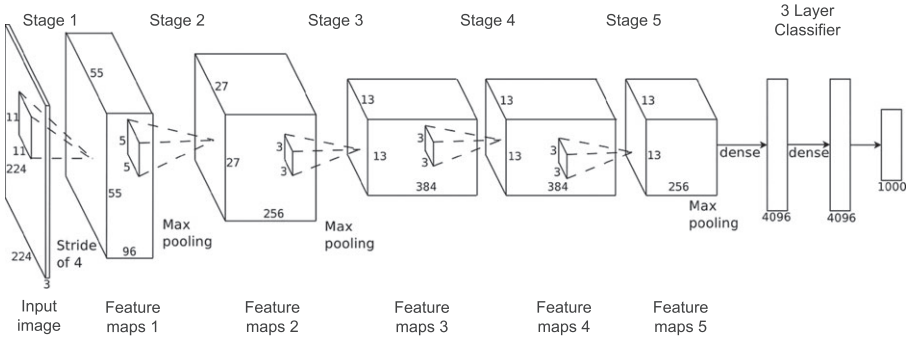

Figure 1. A modern deep convolutional neural network with 832M floating point MAC (multiply accumulate) operations and 60 million network parameters, this network won the ImageNet 2012 contest.

results and makes a simple comparison between our implementation and existing works. Section 5 reports related work. Section 6 concludes the paper.

2. BACKGROUND

2.1. Overview of the convolutional neural network algorithm

Convolutional neural network is a trainable architecture inspired by the research in neuroscience [15]. It has two computational paths, a forward propagation path for classification and a backward propagation path for training. In practice, many applications first train the CNN off-line using high-performance clusters to reduce the training time and then run the trained CNN at job site using dedicated devices. In this paper, we only focus on speeding up the forward path.

A typical CNN structure consists of a feature extractor and a classifier. The feature extractor extracts an input image's features and sends them to the classifier. According to these features, the classifier decides the category that the input image belongs to. A feature extractor consists of several similar stages. The input and output of a stage are called feature maps. The output feature maps of a stage are the input of the next stage. The input image is the input to the first stage. Each stage consists of three layers: a convolutional layer, a non-linearity layer, and a sub-sample layer. The output feature maps of the last stage are organized as a feature vector of the original input image and sent to the classifier. A classifier is a traditional MLP (multi-layer perceptron) composed of several full connection layers. It takes the feature vector as input and calculates the probability of each category that the input image may belong to. At last, the classifier chooses the category with highest probability as the output. Figure 1 shows the structure of a representative deep CNN AlexNet that won the ImageNet 2012 contest [1].

$$Y_r = bias + \sum_{q=0}^{Q-1} conv < X_q, K_{r,q} > \tag{1}$$

$$conv < X_q, K_{r,q} > [m][n] = \tag{2}$$

$$\sum_{w=0}^{Ksize-1} \sum_{l=0}^{Ksize-1} K_{r,q}[w][l] * X_q[m * stride + w][n * stride + l]$$

Most of the workload of CNN exists in the convolutional layers. Figure 2 shows the computational procedure of a convolutional layer. A convolutional layer has Q input feature maps $X_0 \ldots X_{Q-1}$ and R output feature maps $Y_0 \ldots Y_{R-1}$. To produce an output feature map Y_r, all input feature maps $X_0 \ldots X_{Q-1}$ are first individually convolved with convolutional kernels $K_{r,0} \ldots K_{r,Q-1}$. Then, all the Q convolved maps are summed to be one map. At last, a bias value is added to each pixel of the map to get Y_r. Q*R convolutions are performed per convolutional layer. Equations (1) and (2) show the

DOI: 10.1002/cpe

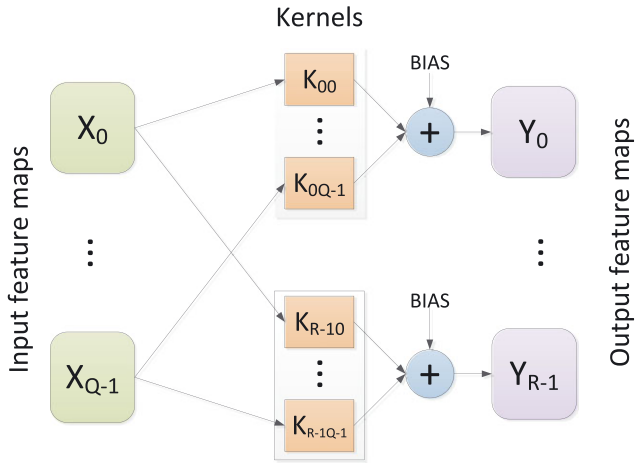

Figure 2. Computational procedure of a convolutional layer.

mathematical form of the procedure. Here, conv$< X_q , K_{r,q} >$ means the convolution between input feature map X_q and convolutional kernel $K_{r,q}$. In (2), K_{size} is the size of the convolutional kernel, and $stride$ means the distance that the convolutional window slides each time.

2.2. *Matrix multiplication accelerators*

Traditional matrix multiplication accelerators [16] adopt the architecture with a linear array of processing elements (PEs). Such a structure divides a matrix multiplication task into several sub-block computational tasks.

The matrix blocking method is described by Equations (3) and (4). In order to get the product matrix C (M x N) of matrix A (M x K) and matrix B (K x N), we need the following steps: first, dividing C to m rows and n columns blocks, as shown in Equation (3) and then computing $C_{i,j}$ ($i = 0, 1, 2 \ldots m-1$, $j = 0, 1, 2 \ldots n-1$) individually. Equation (4) describes the procedure. $A_{i,k}$ ($k = 0, 1, 2 \ldots K-1$) are column vectors, and $B_{j,k}$ ($k = 0, 1, 2 \ldots K-1$) are row vectors. The algorithm finishes after all the $C_{i,j}$ blocks are computed.

$$\begin{aligned} C = AB &= [A_0, A_1, \ldots, A_{m-1}]^T [B_0, B_1, \ldots, B_{n-1}] \\ &= \begin{bmatrix} C_{0,0} & \ldots & C_{0,n-1} \\ \ldots & \ldots & \ldots \\ C_{m-1,0} & \ldots & C_{m-1,n-1} \end{bmatrix} \end{aligned} \tag{3}$$

$$\begin{aligned} C_{i,j} &= A_i B_j \\ &= [A_{i,0}, A_{i,1} \ldots A_{i,K-1}][B_{j,0}, B_{j,1} \ldots B_{j,K-1}]^T \\ &= A_{i,0} * B_{j,0} + A_{i,1} * B_{j,1} + \ldots + A_{i,K-1} * B_{j,K-1} \\ &= C_{i,j}^0 + C_{i,j}^1 + \ldots + C_{i,j}^{K-1} \end{aligned} \tag{4}$$

A classic structure of matrix multiplier [16] is organized as Figure 3 shows. Every time, the matrix multiplier works as described by Equation (4). It gets column vectors $A_{i,k}$ and row vectors $B_{j,k}$ ($k = 0, 1, \cdots, K-1$) to compute a $C_{i,j}$ block. The matrix multiplier consists of a PE chain. $A_{i,k}$ and $B_{j,k}$ are passed through the chain. Each PE every time holds an element of $A_{i,k}$ and multiplies it with all the elements of $B_{j,k}$ to get a row vector of the intermediate result $C_{i,j}^k$ and accumulates it. The FIFO_A and FIFO_B are used to pass the elements of $A_{i,k}$ and $B_{j,k}$ to next PE. The local ports allow local registers RA (working as a double buffer, selector MUX1 chooses the correct data that will be used) and RB to get the required elements. A multiplier and an accumulator form an MAC (multiply accumulate) unit. The multiplier receives the input element from RA and RB. The result of the multiplier is sent to an accumulator to be added with the intermediate result from the

DOI: 10.1002/cpe

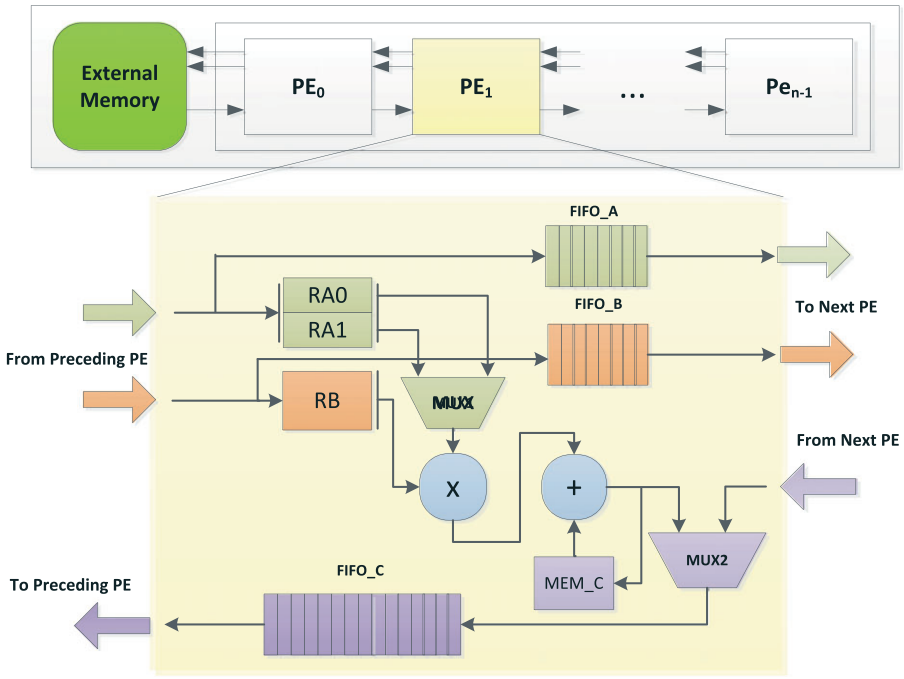

Figure 3. A classic structure of matrix multiplier.

local memory unit MEM_C, and the sum is written back to MEM_C. The address of MEM_C is generated from a local state machine. The final result is written to FIFO_C to be passed to the external memory. The writing back order is determined by a selector MUX2, and the local result must be written back before FIFO_C receives the result from the next PE. For one PE chain, only one element from matrix A and one element from matrix B need to be loaded in one clock cycle. The time of writing back usually can be ignored when K is far larger than the block size.

3. DESIGN AND ARCHITECTURE

The top view of our SoC design with a CNN accelerator is given in Figure 4. The host processor communicates with the accelerator through a system bus (It can be an AXI bus for a typical ARM SoC or a Wishbone bus for an Open RISC SoC) and handles the workload except for the convolutional layers and full connection layers. The accelerator and the host processor share the external memory (the main memory of the host processor, the type is DRAM memory). The accelerator consists of several chains. Each chain has a stream-prefetcher, a stream-mapper, and a matrix multiplier. The matrix multiplier accelerates the matrix multiplication workload existing in the convolutional layers and the full connection layers. It consists of a stream S/L (Store/Load) unit and hundreds of PEs (processing elements). The stream S/L unit loads operands to the PE chain (computational unit of the matrix multiplier) and then stores the results. The stream-mapper remaps the data stream to the stream S/L unit to unroll convolutions to matrix multiplications. The stream-prefetcher is used to ensure efficient external memory access.

3.1. Using stream-mapper to unroll convolutions to matrix multiplications

To utilize the efficient matrix multiplier structure, we need to unroll the convolutions to matrix multiplications first. Figure 5 shows a simple example of unrolling the convolutions of one convolutional layer to a matrix multiplication. (A) is the normal computational procedure of the convolutional layer as the last section introduced. The input feature maps of this layer are X_0, X_1, X_2, and the output feature maps of this layer are Y_0, Y_1. There are $R * Q = 6$ convolutional kernels in this layer. (B) shows how to unroll the convolutions in this layer to a matrix multiplication. Input feature

DOI: 10.1002/cpe

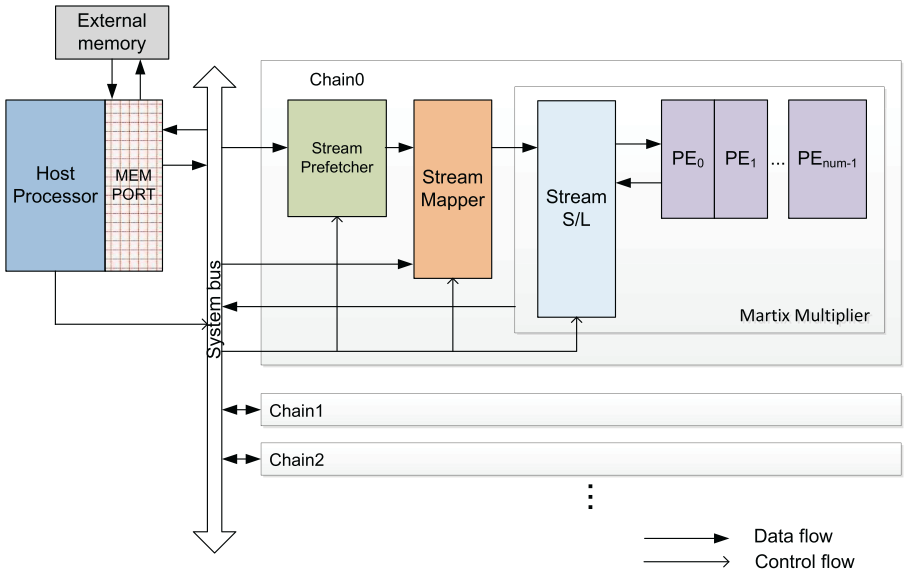

Figure 4. Top structure of our design.

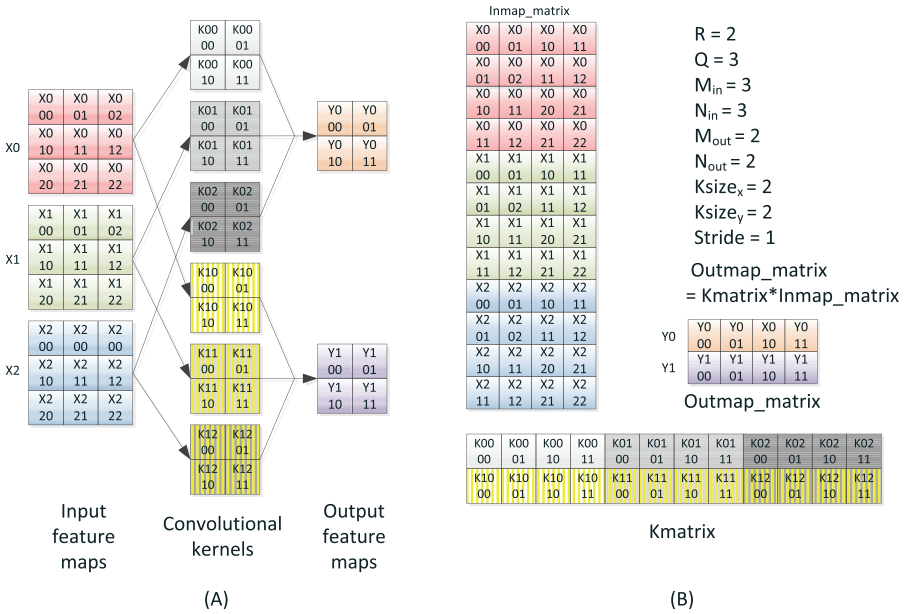

Figure 5. An example of how to unroll convolutions to a matrix multiplication.

maps, convolutional kernels, output feature maps are organized as Inmap_matrix, Kmatrix, and Outmap_matrix, respectively. Because a two-dimensional matrix is stored as an array in the memory, both Kmatrix and Outmap_matrix keep their data forms unchanged. Only Inmap_matrix needs to be reorganized. There are four 2×2 convolutional windows in every input feature map. Each of them is organized as a four-length column vector, and the four vectors are combined as a 4×4 matrix block. For the three input feature maps, we obtain three matrix blocks, and then, we pile the three blocks up to obtain the 12×4 Inmap_matrix. After multiplying Kmatrix with Inmap_matrix, we obtain the Outmap_matrix, and the result is as the same as procedure (A).

Using software to rearrange the data first is time-consuming and requires extra memory space. We run Caffe with AlexNet on a CPU. Figure 6 shows the unrolling overhead of its five convolutional

DOI: 10.1002/cpe

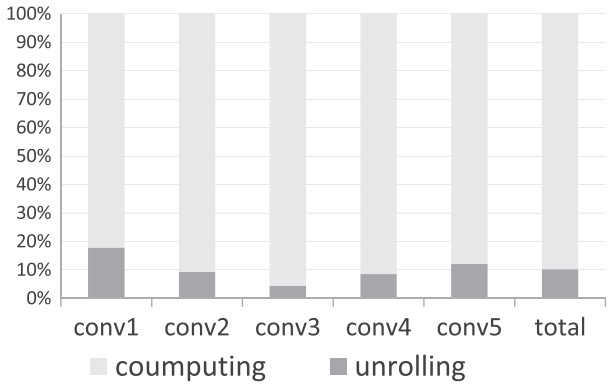

Figure 6. The overhead of unrolling on CPU using Caffe AlexNet test case.

```
input: Bx,By
parameter:
        Ksize, win_num, stride, image_size, img_addr
behavior:
ofs_cwin_y = Bx % Ksize;
              //column offset in a convolutional window
temp = Bx / Ksize;
im_num = temp / Ksize;
              //which input feature map is
ofs_cwin_x = temp % Ksize;
              //row offset in a convolutional window
cwin_x = By / win_num;
cwin_y = By % win_num;
   //convolutional window offset in a input feature map
img_x = cwin_x * stride + ofs_cwin_x;
img_y = cwin_y * stride + ofs_cwin_y;
ofs_pix = img_x * img_size + img_y;
              //pixel offset in a input feature map
ofs_im = im_num * img_size^2;
              //feature map offset
real_addr = img_addr + ofs_im + ofs_pix;
              //get the offset in input feature maps
```

Figure 7. The pseudo-code of the mapping method.

layers. About 10% of the total time is consumed in average. If we only use a matrix multiplier to accelerate the computational part and still leave the data rearrangement part on the host processor, according to Amdahl's law, the upper bound of speed-up is only 10×.

A hardware stream-mapper is designed to eliminate the overhead of unrolling convolutions to matrix multiplications. The operation (*KmatrixxInmap_matrix* = *Outmap_matrix*) is executed on the matrix multiplier. The matrix multiplier is driven by the data stream from stream S/L unit, which loads elements of Kmatrix and Inmap_matrix to the PE chain and stores the result. A stream-mapper is placed between the stream S/L and the system bus. All the data access to the elements of Inmap_matrix are directly mapped to the elements of the input feature maps. Thus, Inmap_matrix does not occupy any memory space. The data are only stored in the form of input feature maps.

The mapping algorithm is given in Figure 7. The input is the location of the element Inmap_matrix[Bx,By] that the stream S/L attempts to access, and the output is the address of the Inmap_matrix[Bx,By] in input feature maps which are stored in the external memory. All the parameters needed to compute the address are set by the host processor. Ksize denotes the convolutional kernel size, and win_num is the number of convolutional windows in one dimension of an input

DOI: 10.1002/cpe

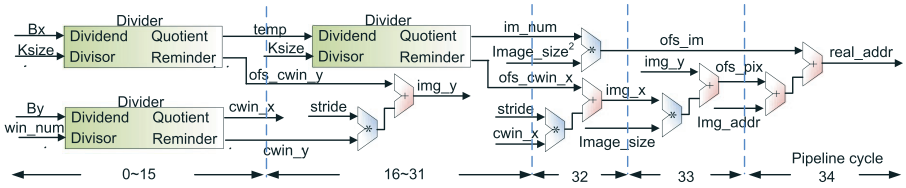

Figure 8. The structure of the stream-mapper.

feature map. The image_size parameter is the number of pixels in one dimension of an input feature map. The img_addr is the address of the first feature map's address.

To get the address, it first calculates the element's offset in a convolutional window (ofs_cwin_x, ofs_cwin_y). Then, it calculates the offset of the convolutional window in a input feature map (cwin_x, cwin_y). Thereafter, it can calculate the pixel (the required element) offset in the feature map (ofs_pix), followed by calculating which input feature map the element belongs to (im_num), and calculating the feature map's offset (ofs_im). Because the first feature map's address (img_addr) is known, the mapper can get the address of the element by adding all the offsets to img_addr.

The structure of the stream-mapper is shown in Figure 8. It is a logic unit to implement the process of the mapping algorithm. The stream-mapper receives one element location from stream S/L and generates one address every clock cycle. The addresses are generated as a stream. It is time-consuming to map the address in one clock cycle, so a 35-level pipeline is used. Three divisions are the most time-consuming parts of the mapper, and each of them needs 16 cycles to obtain one result. Although we overlap two of the divider, there are still 32 cycles left. Some stack registers for synchronizing are not shown in the figure for simplicity. Most of them exist where the blue dash lines are.

$$Conv_{times} = \left(\frac{Img\_size - Ksize}{stride} + 1\right)^2 \tag{5}$$

$$Size_{inmap} = Img\_size^2 * Q \tag{6}$$

$$Size_{inmap\_matrix} = Ksize^2 * Q * Conv_{times} \tag{7}$$

$$MA_{inmap\_soft} = Size_{inmap} + Size_{inmap\_matrix} * (1 + \alpha) \tag{8}$$

$$MA_{inmap\_hard} = Size_{inmap\_matrix} * \alpha \tag{9}$$

$$MA_{kmatrix} = Size_{kmatrix} * \beta = (Ksize^2 * Q * R) * \beta \tag{10}$$

$$MA_{outmap} = \left(\frac{Img\_size\text{-}Ksize}{stride} + 1\right)^2 * R \tag{11}$$

$$MA_{soft} = MA_{inmap\_soft} + MA_{kmatrix} + MA_{outmap} \tag{12}$$

$$MA_{hard} = MA_{inmap\_hard} + MA_{kmatrix} + MA_{outmap} \tag{13}$$

Equations (5) $\sim$ (13) quantify the memory access times of software and our hardware data rearrangement, respectively. Software rearrangement needs two memory areas. One for the input feature maps whose size is $Size_{inmap}$, the other is for the Inmap_matrix whose size is $Size_{inmap\_matrix}$. The host processor first reads the elements from the input feature maps and then writes them to the Inmap_matrix. After the data are organized as the matrix form, the host processor starts the multiplier. The multiplier accesses Inmap_matrix for α times and Kmatrix for β times, the value of α and β are determined by the matrix size and the matrix blocking method. The hardware implementation only needs the memory area for input feature maps. It maps all the access to the Inmap_matrix

DOI: 10.1002/cpe

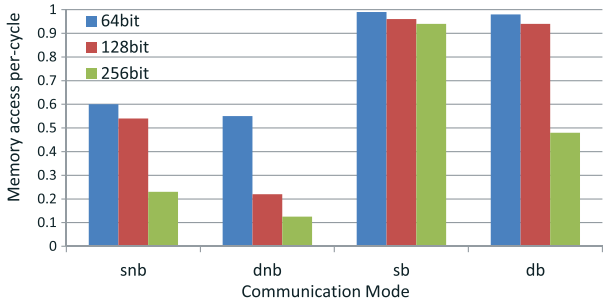

Figure 9. Memory access per cycle for different communicate mode.

to the input feature maps directly. Compared with a software implementation, a memory space of $Size_{inmap\_matrix}$ and $Size_{inmap} + Size_{inmap\_matrix}$ times memory access are eliminated, and the data rearrangement and the matrix multiplication can be overlapped completely.

3.2. A prefetch strategy to make the address stream sequential

The address stream from the stream-mapper to external memory is not sequential, which prohibits the incremental addressing burst characteristic of the memory port (A burst memory transaction processes multiple memory access in a pipelined way which can take full use of the memory bandwidth). It will significantly reduce the memory bandwidth utilization. Figure 9 shows a simple memory test in an FPGA platform with DDR3 memory. The physical memory bandwidth is 1066 MHz, 32-bit and the frequency of memory requests is 150 MHz. The horizontal axis presents the communication mode, s means simplex, d means duplex, nb means non-sequential address stream without burst transmission, and b means sequential address stream with burst transmission. The vertical axis presents the memory access per cycle. Obviously, memory access with the burst characteristic performs much better than the non-burst access.

To keep the address stream sequential, we prefetch the data as a sequential way. A novel prefetch strategy for CNN is shown in Figure 10, taking the situation that the $Ksize = 3$, the $Img\_size = 8$ and the $stride = 1$ as a example. There are two input feature maps in this example, and the length of the PE chain is 8. The original memory access order is given in Figure 10(a). Once a time, the stream S/L loads PE_NUM elements from the Inmap_matrix as a row vector. Although the location stream of the Inmap_matrix elements is sequential, the mapped addresses from the stream mapper are not sequential. We divide the Inmap_matrix into two parts as shown in Figure 10(b), the upper part consists of the elements from X0, and the lower part consists of the elements from X1. Back to Figure 10(a), the stream of the row vectors is divided to blocks from different input feature maps. From Figure 10(c), we find that, according to the map algorithm, all the elements from a block can be located in a sequential area in one input feature map. We prefetch these sequential areas to an on-chip prefetch buffer before the matrix multiplier accesses them. Figure 10(b) shows the first four blocks' prefetch areas. Two prefetch parameters need to be determined. They are the start address and prefetch length. The start address is location of the first element of one block. It is time-consuming to get the ideal prefetch length, because of the variable network structure. We have used an approximate value shown in Equation (14), which is a little bigger than the ideal one. In most cases, the prefetch area is smaller than the block, whose size is $PE\_NUM * Ksize^2$, especially when $stride = 1$.

A prefetch unit has been designed to overlap the prefetch and computing. We use the double buffering technique. When the stream S/L unit is loading a block from one prefetch buffer, the next prefetch area is being loaded into the other prefetch buffer. The address stream becomes sequential, and the number of memory access decreases after the prefetch unit is integrated.

$$Length_{prefetch} = \left(\left\lceil \frac{Block\_size}{win\_num} \right\rceil * stride + Ksize - 1 \right) * Img\_size + Ksize \tag{14}$$

DOI: 10.1002/cpe

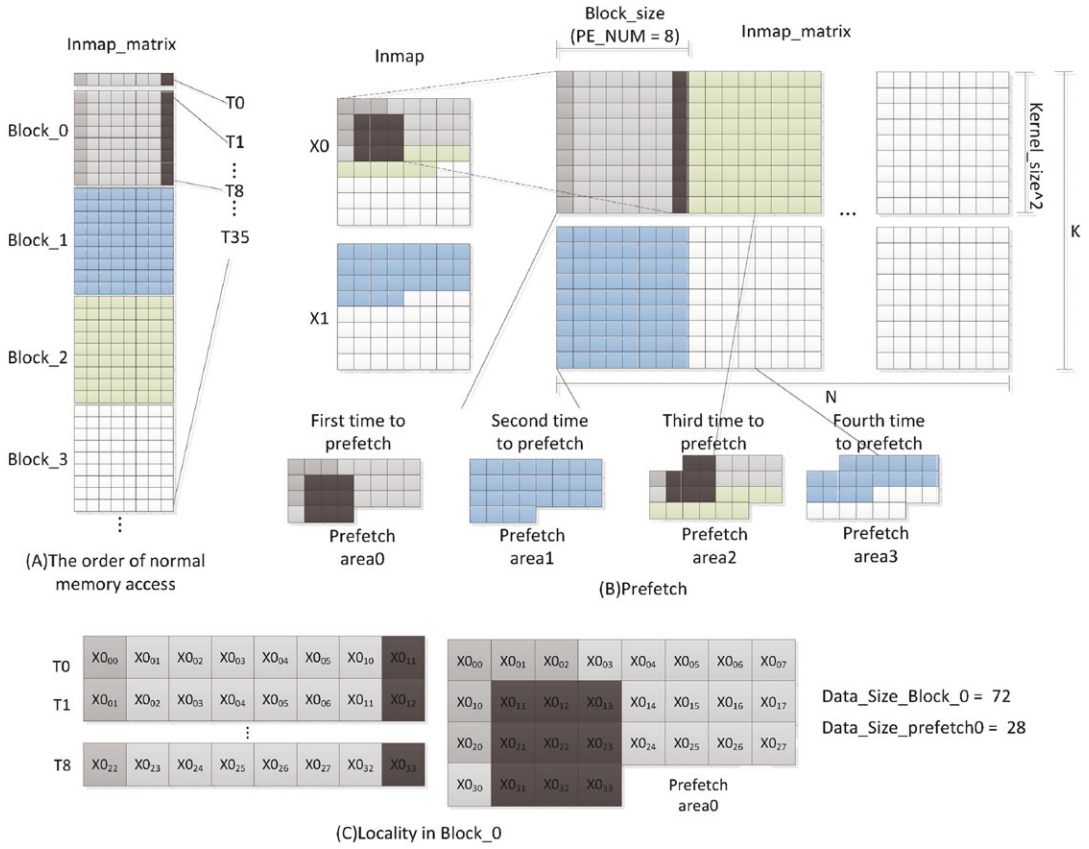

Figure 10. The procedure of prefetching.

3.3. Accelerating matrix multiplications of different sizes

Most of the computational workload now can be formulated as matrix multiplications. The matrices unrolled by different convolutional layers may differ greatly in size. Mapping such different sized matrix multiplications efficiently to our accelerator has two challenges. The first is the scale of the matrix being too small to be allocated to multiple chains or even PEs in one chain. This problem limits the parallel potential of the network structure. Although processing multiple data at the same time may help, this paper does not focus on this approach. The other problem is that some dimensions of the matrices are too small to obtain appropriate sub-blocks to take full use of the PEs of one chain.

As shown in Figure 11, in general, Matrices A and B for example can be divided into four sub-matrices, where ① and ③ are sub-matrices with sizes that are multiples of the number of PEs in a chain. Thus, the computation can be divided into four parts: ① × ③, ① × ④, ② × ③ and ② × ④ ($M > S_i > S'_i, N > S_j > S'_j$). According to the matrix multiplier mentioned in Section 2, when the size of sub-blocks is equal to the number of PEs in a chain (S_i), all PEs are used. When $S'_i < S_i$, only S'_i PEs can be used. The computation of ② × ③ and ② × ④ may affect the computational efficiency of the total matrix. If M is large enough, the affect can be ignored, but for many unrolled matrices in a CNN, the efficiency drop is obvious.

The blocking strategy can be improved to alleviate the problem. First, we construct an objective function to calculate the computation time of the workload based on a single chain structure. $T_{1,3}$, $T_{1,4}$, $T_{2,3}$, $T_{2,4}$ are the computation cycles of ① × ③, ① × ④, ② × ③ and ② × ④, respectively. The objective function is the sum of $T_{1,3}$, $T_{1,4}$, $T_{2,3}$, $T_{2,4}$ because the matrix multiplier calculates the four parts sequentially.

$$f(S_i, S_j) = T_{1,3} + T_{1,4} + T_{2,3} + T_{2,4} \tag{15}$$

DOI: 10.1002/cpe

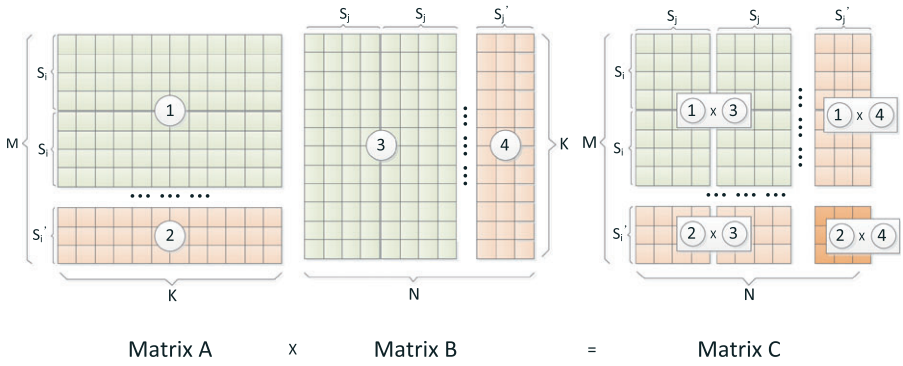

Figure 11. An illustration of non-uniform matrices blocking

The algorithm needs to prefetch S_i elements of matrix $\mathbf{A}$ at first, which takes S_i cycles. Thus, the setup time of the PE chain is S_i cycles. Each element of $\mathbf{A}$ is used S_j times. When $S_i \neq S_j$, according to the algorithm, it takes $\max\{S_i, S_j\} \times K$ cycles to calculate a sub-block. We assume $m = \lfloor M/S_i \rfloor$, $n = \lfloor N/S_j \rfloor$, so a $S_i \times S_j$ sub-block computation requires $S_i + \max\{S_i, S_j\} \times K$ cycles, thus the total computation cycles of ① × ③ is

$$T_{1,3} = m \times n \times \left(S_i + \max\left\{S_i, S_j\right\} \times K\right) \tag{16}$$

Because the prefetch time of the first column of each $S_i \times K$ sub-block can be ignored when compared with the computation time, we simplify Equation (16) as

$$T_{1,3} = m \times n \times \left(\max\left\{S_i, S_j\right\} \times K\right) \tag{17}$$

Denote $S_i' = M - S_i \times m$, $S_j' = R - S_j \times n$. In a similar way, we obtain

$$T_{1,4} = k_2 \times m \times \left(\max\left\{S_i, S_j'\right\} \times K\right) \tag{18}$$

$$T_{2,3} = k_1 \times n \times \left(\max\left\{S_i', S_j\right\} \times K\right) \tag{19}$$

$$T_{2,4} = k_1 \times k_2 \times \left(\max\left\{S_i', S_j'\right\} \times K\right) \tag{20}$$

The values of k_1 and k_2 follow the following relations:

$$k_1 = \begin{cases} 0, & \text{M is multiplies of } S_i \\ 1, & \text{M is not multiplies of } S_i \end{cases} \tag{21}$$

$$k_2 = \begin{cases} 0, & \text{N is multiplies of } S_j \\ 1, & \text{N is not multiplies of } S_j \end{cases} \tag{22}$$

There are constraints on the values of S_i and S_j. First, the value of Si can not be greater than the number of PEs according to the blocking algorithm. Second, the data hazards in pipeline should be taken into consideration, which occurs when data required for an MAC operation are delayed in the addition pipeline. Therefore, we have

$$\begin{cases} S_i \leqslant P \\ \max\{S_i, S_j\} \geqslant Stage_{add} \\ \max\{M - S_i \times m, N - S_j \times n\} \geqslant Stage_{add} \end{cases} \tag{23}$$

$Stage_{add}$ is the needed number of additional stages to avoid data hazards. P is the number of PEs in a chain. To simplify the discussion, we assume $S_i = S_j$ and $S_i' > S_j'$; therefore, the following conclusion holds:

DOI: 10.1002/cpe

When $k_1 = 0, k_2 = 0$;
Because $m = M/S_i, n = N/S_j$, we have

$$f(S_i, S_j) = T_{1,3} = F(S_i) = \frac{M \times N \times K}{S_i} \tag{24}$$

When $k_1 = 0, k_2 = 1$;
Because $m = M/S_i, n = \lfloor N/S_j \rfloor$, we therefore obtain

$$f(S_i, S_j) = T_{1,3} + T_{1,4} = F(S_i) = M \times K \times \lfloor N/S_i \rfloor + M \times K \tag{25}$$

When $k_1 = 1, k_2 = 0$;
Similar to case 2, because $m = \lfloor M/S_i \rfloor, n = R/S_j$, the following holds:

$$f(S_i, S_j) = T_{1,3} + T_{2,3} = F(S_i) = \lfloor M/S_i \rfloor \times N \times K + N \times K \tag{26}$$

When $k_1 = 1, k_2 = 1$;

$$F(S_i) = S_i \times (\lfloor M/S_i \rfloor \times \lfloor N/S_i \rfloor + \lfloor N/S_i \rfloor) \times K + M \times K \tag{27}$$

When k_1 and k_2 are both 0, it can be seen that the optimal block size is equal to P according to Equations (24)(25). When k_1 and k_2 are not simultaneously 0, we can, for example, use MATLAB to find the minimum value of $f(S_i, S_j)$.

For the reason that all the sub-blocks' optimized size are the same, we directly parallelize the sub-blocks to the multi-chain structure.

3.4. Batch processing for full connection layers

In the full connection layers, the main computational type is matrix-vector multiplication. The ratio of computation/memory access of a matrix-vector multiplication is low. Every element in parameter matrix W only does one multiply accumulate operation. Thus, the computational resources can not be fully used because of the limited memory bandwidth. We compute multiple images' full connection layers together that merges a batch of matrix-vector multiplications into a matrix–matrix multiplication as Figure 12 shows. Every element in matrix W needs to be multiplied with batch size

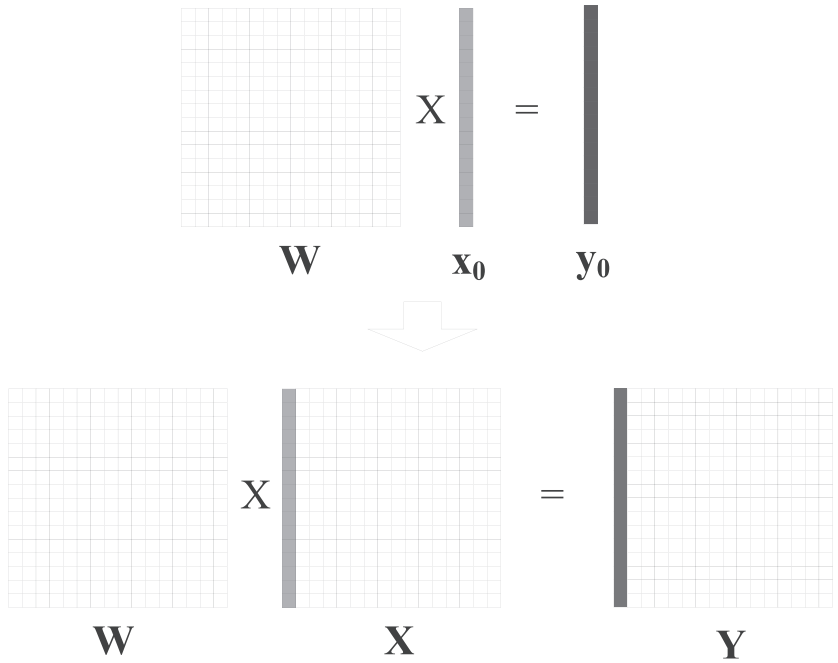

Figure 12. Merging matrix-vector multiplications to a matrix–matrix multiplication.

DOI: 10.1002/cpe

elements in X. The ratio of computation/memory access increases when the batch size increases. We bypass the stream-mapper and prefetch unit in the full connection layers. In such design, the full connection layers share the matrix multiplier with the convolutional layers, which saves computational resources and improves the efficiency.

3.5. Supporting the software framework Caffe

To ease the implementation, we make use of a popular CNN framework Caffe. The implementation of the convolutional layer and full connection layer of Caffe are replaced, keeping the interfaces to all other parts of the framework unchanged. All the implementation details are transparent to the user; thus, they can use Caffe as usual. To change the network structure, they only need to modify the Caffe's configure file.

For a traditional accelerator (like GPU), the host processor first copies the data to be processed to a dedicated memory device (like video memory) by DMA. After the data are processed by the accelerator, they are copied back to the host processor's main memory. For our case, we only offload the convolutional layer and the full connection layer to the accelerator and leave the other parts of the Caffe framework on the host processor. If we use the traditional way, the overhead of frequent data copies is unacceptable. It is a better choice to make the host processer and the accelerator to work in the same memory space.

Although the main workload of Caffe is offloaded to the accelerator, the framework is still running on the host processor. Caffe's framework relies on many libraries of the operating system (Linux). It accesses DRAM through the operating system's user space. The user space is a virtual space that is mapped to some discontinuous physical memory pages. The mapping from the user space to physical space is managed by the host processor's memory management unit (MMU). In today's FPGA-based SoC, the host processor exists as an IP core, and it is hard for the accelerator to use the host processor's MMU to access the user space. Most accelerators can only access DRAM through the physical space, which is continuous, so Caffe needs to allocate continuous physical space for the accelerator. Two problems exist. First, the maximum size the operating system can allocate for continuous physical space is limited in Linux (only several MB) [17], which can not satisfy the Caffe's requirement (at least hundreds of MB). Second, the continuous physical space only can be allocated in the kernel space in Linux, an extra data copy overhead between the user space and the kernel space is needed. We thus propose a unified virtual memory support mechanism to solve these two problems.

Unlike the traditional accelerator memory access mechanism that allocates continuous physical space in the kernel space as Figure 13(a) shows, we leave enough continuous physical memory space as device memory space in the Linux boot time. Such device memory space can be based on dedicated memory devices of the FPGA or just a part of the main memory device of the host processor. The space can be accessed by both the host processor and the accelerator using physical address. All the data needed to be processed by the accelerator are allocated in the device memory space. Although the accelerator can not use MMU to access host processor's user space, the host

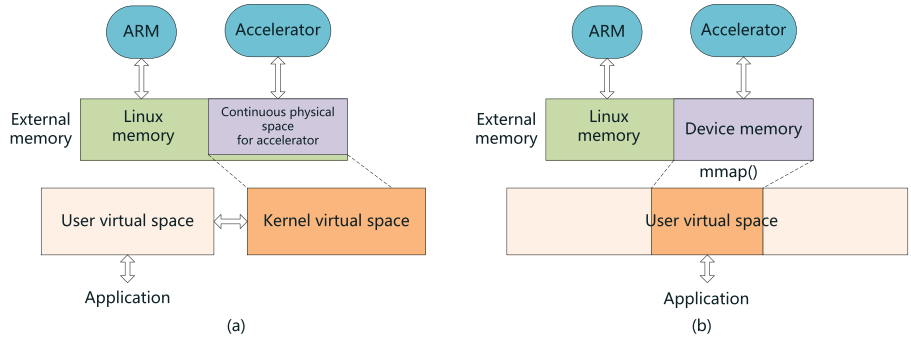

Figure 13. (a) Traditional accelerator memory access mechanism and (b) unified virtual memory support mechanism.

DOI: 10.1002/cpe

Table I. This table describes the environment on one chip, there are two chips in the prototype system.

Item	Content
Chip	Xilinx Zynq-zq7045
Host processor	Dual core ARM cortexA9 800 MHz
L1 Cache	32 KB D-Cache 32 KB I-Cache
L2 Cache	512 KB
Main memory	1GB DDR3 1066 MHz 32 bit
OS	PetaLinux(kernel visionč̌3.14)
FPGA	Kintex-7
Accelerator frequency	150 MHz
Benchmark	Caffe

Table II. This table describes the specification of the general purpose devices.

Type	Device
CPU	Intel Xeon X5675 6-core 3060-3460 MHz
GPU	Nvidia Tesla K20

processor uses the MMU to map a part of user space to the device memory space, which can be achieved by using Linux system function mmap(). Figure 13(b) shows our implementation. In this way, the continuous physical space for accelerator is large enough, and the framework can access the device memory in the user space. There is no extra copy overhead anymore. We discuss the implementation details of our memory management scheme in [18].

4. EXPERIMENTAL EVALUATION

4.1. Evaluation methodology

To evaluate our design, we have implemented a prototype system, which is based on two Xilinx Zynq-zq7045 FPGA SoC chips. The experimental environment is summarized in Table I. The accelerator is implemented in the FPGA part of the Zynq chip working at 150 MHz. As a baseline software implementation, we first port Caffe to the host processor, which is a dual core ARM-cotexA9 integrated in the Zynq chip, working at 800 MHz. Then, we offload the workload of convolutional layers and full connection layers from the host processor to the accelerator. Data parallelism is implemented between the two chips, and the only communication between the two chips is for collecting the results via the Ethernet. We first evaluate the impact of the stream-mapper and stream-prefetcher and test the performance of the accelerator for different CNN structures. Then, we make a comparison between our design and existing FPGA-based CNN accelerators. At last, we also compare with general purpose devices such as a high-end GPU and a high-end CPU. The specification of the general purpose devices is shown in Table II.

4.2. The impact of stream-mapper and stream-prefetcher

Figure 14 shows the accelerator performance of the structure with only a matrix multiplier on AlexNet. The columns denoted by 'soft' refer to the soft implementation running at the host processor. For five convolutional layers (conv1 $\sim$ conv5), we use the host processor to rearrange the input feature maps data to Inmap_matrix. Different accelerator sizes are tested, 50*2 means that there are two accelerator chains in each chip and each of them has 50 PEs. The result shows that the performance does not improve noticeably when the PE number or the number of the chains increases. For conv1, the speed-up compared with the software implementation is even less than 10$\times$ for all accelerator scale sizes. The reason is the software unrolling overhead. As shown in Figure 6, the unrolling overhead of conv1 is more than 18%. It means that the maximum achievable speed-up is 5.5$\times$. As we use two host processors to parallelize the task, the maximum speed-up increases to

DOI: 10.1002/cpe

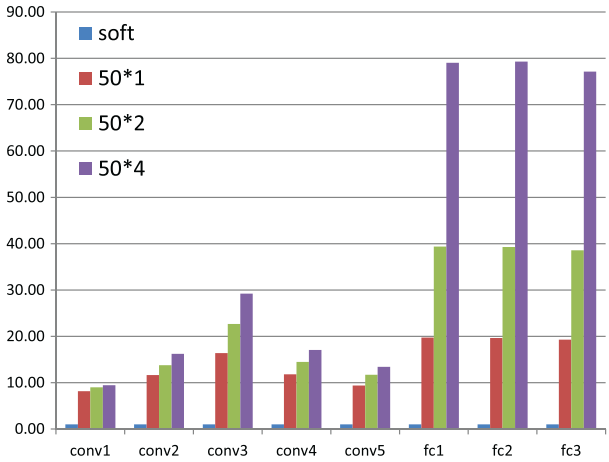

Figure 14. Performance without stream-mapper and prefetch unit.

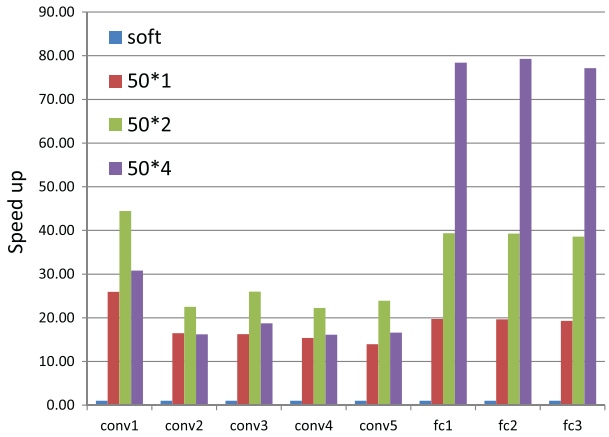

Figure 15. Performance with stream-mapper but without prefetch unit.

11. The problem does not exist in the three full connection layers (fc1 $\sim$ fc3). There is no software data rearrangement part in these layers, and they use the matrix multiplier directly. For this reason, speed-up of the full connection layers is much higher than the convolutional layers.

Figure 15 shows the accelerator performance with the stream-mapper but without the prefetch unit. Although an obvious improvement is achieved compared with the bare matrix multiplier, there is a big difference from the speed-up of the full connection layers. Because of the non-sequential external memory address stream from the stream-mapper, the memory requests are discrete, and the burst characteristic of the memory can not be used. It causes low bandwidth utilization and decreases the performance. The phenomenon is serious when all the four chains are used. The frequent memory access conflicts even make the performance poorer than using two chains.

Figure 16 shows the accelerator performance with both a stream-mapper and a prefetch unit. The time overhead of unrolling is eliminated, and the memory burst characteristic can be used. Compared with the former two structures, the performance of the convolutional layers is improved dramatically. For the reason that the software edition has extra overhead of unrolling, the convolutional layers obtain an even higher speed-up than the full connection layers. This is most obvious in conv1 whose software unrolling overhead is the highest, where a speed-up of more than 160$\times$ is achieved.

4.3. Performance portability of different convolutional neural network structures

We have used three different CNN structures to evaluate the performance portability of our accelerator. Figure 17 shows the accelerator performance of different CNN structures. The largest chain

DOI: 10.1002/cpe

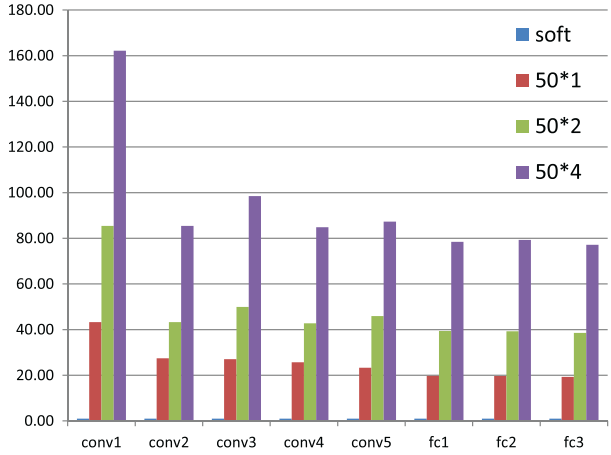

Figure 16. Performance with stream-mapper and prefetch unit.

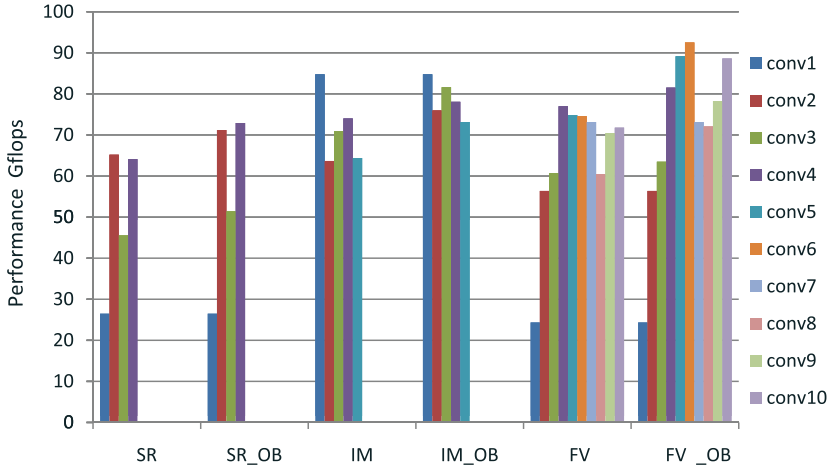

Figure 17. Performance of different convolutional neural network structures.

size is 4*50 for each chip. Three different CNNs structures are tested. The first one is a four convolutional layer structure for street scene recognition (SR) [19]. The second one is the AlexNet for ImageNet classification (IM). The third one is a 10 convolutional layer structure for face verification (FV) [20]. For each structure, we give two group results; one uses our optimized blocking (OB) method, and one does not.

The sizes of the unrolled matrices for all the convolutional layers are shown in Table III. It can be observed that some layers perform poorly. The performance is decided by the scale of each convolutional layer rather than the number of layers of the whole network. FV has the deepest network, but its first convolutional layer is too small to utilize the accelerator efficiently. There are two reasons for the performance decline. One is the layer's scale is too small to be distributed to multiple chains, such as in conv3 and conv4 of SR, only 2 or 3 chains can be used. The other is some dimensions of the matrices are too small to obtain appropriate sub-blocks to take full use of the PEs of one chain. As mentioned in Section 3, our optimized block method is meant to reduce the negative effect from the second reason. Table IV shows the OB size for each layer. The results show that after using our blocking strategy, the performance is improved obviously. However, in some extreme cases, the OB strategy does not work well. In conv1 of SR, the dimension M is only 12, which is far smaller than the chain size 50 (one full block size), while the dimension N is far bigger than the chain size. It means, in dimension M, the blocking strategy does not work, and in dimension N, the incomplete block almost will not affect the performance. In conv1 of FV, the dimension K

DOI: 10.1002/cpe

Table III. Unrolled matrices sized for all convolutional layers of the three convolutional neural network structures.

	SR			IM			FV		
Net	M	N	K	M	N	K	M	N	K
conv1	12	7056	276	96	3025	363	32	10,000	27
conv2	32	1156	972	128	729	1200	64	10,000	288
conv3	48	81	2592	384	169	2304	64	2500	576
conv4	128	50	3888	192	169	1728	128	2500	576
conv5	–	–	–	128	169	1728	96	625	1152
conv6	–	–	–	–	–	–	192	625	884
conv7	–	–	–	–	–	–	128	169	1728
conv8	–	–	–	–	–	–	256	169	1152
conv9	–	–	–	–	–	–	160	49	2304
conv10	–	–	–	–	–	–	320	49	1440

SR, scene recognition; IM, ImageNet classification; FV, face verification.

Table IV. Optimized blocking size.

Net	SR	IM	FV
conv1	12	49	32
conv2	32	49	41
conv3	48	50	41
conv4	43	49	44
conv5	–	43	49
conv6	–	–	49
conv7	–	–	43
conv8	–	–	43
conv9	–	–	46
conv10	–	–	46

SR, scene recognition; IM, ImageNet classification; FV, face verification.

Table V. Comparison between our design and existing float-point FPGA-based CNN accelerators.

	FPGA2015[10]	MS CNN [5]	Our design
Precision	32 bits float	32 bits float	32 bits float
Frequency	100 MHz	–	150 MHz
FPGA chip	Virtex7 VX485T	Stratix V D5	Zynq7000 XC7Z045 × 2
DSP available	2800	3180	900 × 2
DSP used	2240	–	800 × 2
CNN type	AlexNet	AlexNet	AlexNet
Real performance	61.62 GFLOPS	182 GFLOPS	77.8 GFLOPS
Performance density	0.022 GOPS/DSP	0.057 GOPS/DSP	0.043 GOPS/DSP

CNN, Convolutional Neural Network; FPGA, Field Programmable Gate Array.

is too small with respect to the computational pipeline, so the start-up costs can not be hidden. In conv1 of IM, the original blocking method works efficiently enough.

4.4. Comparison with other Field Programmable Gate Array Convolutional Neural Network accelerator designs

A comparison between our design and existing FPGA-based float-point CNN accelerators is shown in Table V. The on-chip resources are fully used by our accelerator prototype system as shown in Table VI. The comparison shows that our accelerator could achieve a comparable performance as the state-of-the-art CNN accelerator design. In [5], the authors show the fastest floating-point design

DOI: 10.1002/cpe

Table VI. Critical resource utilization rate in one chip.

Resource	DSP48E	BRAM	LUT	FF
Available	900	545	218,600	437,200
Used	836	402	183,190	218,296
Utilization(%)	92.8	73.8	83.8	49.93

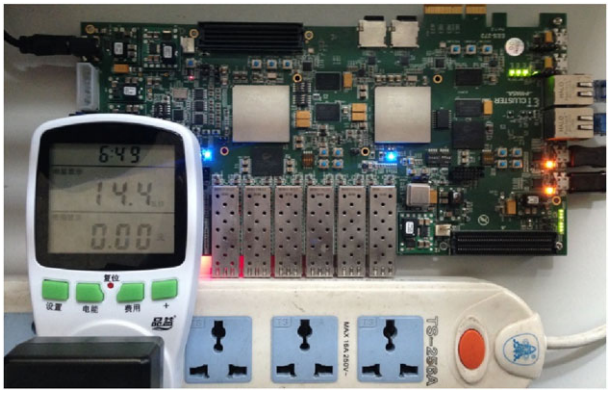

Figure 18. Power consumption of the prototype system.

up to date, and our accelerator gets a comparable performance density using the GOPS/DSP as the performance metric (Design [5] uses a high-end device that can achieve higher work frequency than ours. Because the work frequency is not revealed in [5], we can not provide a more detailed comparison in this paper). The architecture shown in article [10] can only handle the convolutional layer. Extra on-chip resources are needed to build an accelerating module of the full connection layer. Our matrix multiplier-based accelerator adopts the architecture using a linear array of PEs. Modules in such a structure are only connected with their neighbor modules. Almost no global connection is needed. Thus, our accelerator can achieve a relatively high-work frequency.

The author of [4] shows a typical highly customized structure that uses fixed-point PEs and a fixed-size convolution engine. It is three times faster than ours using only 1 Zynq 7045 chip. The first reason is that building a 16-bit fixed point PE only needs two DSPs in today's FPGA, while a 32-bit floating point PE needs four DSPs. A fixed-point computation engine needs fewer DSP units and less on-chip memory resources but also reduces the computational precision. We choose the floating-point data presentation for supporting Caffe. Most of the state-of-the-art trained CNN modules are presented in the floating-point format, so our design can use them directly. The second reason is that [4] only works efficiently for CNNs whose convolutional kernel size is 10×10. Fixed structure is very inflexible.

4.5. Comparison with the general purpose device

As shown in Figure 18, the power consumption of the board is measured using a watt-meter. The comparison between our CNN FPGA accelerator and high-end CPU and GPGPU implementations is shown in Table VII. From the result, we can see that the Intel Xeon X5675 (3.4 Ghz) CPU is inferior in both performance and energy efficiency. Only 22 GFLOPS can be achieved. Only 0.23 Gflops can be provided per watt. The GPGPU provides the highest performance while achieving an energy efficiency of 2.07 Gflops/W. Although our FPGA implementation does not provide a comparable performance with GPGPU, it provides the highest energy efficiency of 5.4 Gflops/W. Actually, the PCB board we used to build the prototype is not customized for this application, so extra energy is wasted on unused device. If we only consider the power consumption of the two FPGA chips (8W), the energy efficiency is 9.725 Gflops/W, which is 4.7× better than a GPU. With respect to performance, a speedup of 3.54× to a CPU implementation with MKL (Intel Math Kernel Library) is achieved.

DOI: 10.1002/cpe

Table VII. Comparison between our design and high-end CPU and GPGPU.

Device	Performance (Gflops)	Power (W)
Intel Xeon X5675	22	95
Nvidia Tesla K20	486	235
Xilinx Zynq FPGA	77.8	14.4

5. RELATED WORK

Many CNN accelerators have been previously proposed. The authors of [9] propose a CNP structure based on a fixed-size convolver. However, it can not efficiently handle convolutions with different kernel sizes. The design presented in [21] gives an improved CNP, the accelerator structure can be reconfigured dynamically to adapt different net structures and multiple feature maps can be processed in parallel, but the size of convolver is still fixed. The authors of [11] propose an accelerator for several machine learning algorithms. The design is also based on a matrix multiplier, but it needs to store all the network parameters on-chip, for a deep CNN, it is unacceptable. And for CNN, it needs to unroll the convolutions to matrix multiplications first by the host processor. The overhead can not be ignored. The authors of [10, 22] make efforts to balance the communication overhead and computational of convolutional layers, but they have not discussed the full connection layers. In fact, the convolutional layers are typically compute-bound, while the full connection layers are memory-bound. The authors of [23] use an ASIC implementation of CNN, it also needs high external-memory bandwidth to support full connection layers.

Implementations based on GPU [13, 24] adapt the convolutions to the matrix multiplications method and obtain the state-of-the-art performance. These works enlighten us to rethink the CNN accelerator architecture. However, GPU suffers from relatively high-energy consumption compared with a FPGA design. Our design also benefits from [16], where the authors have used FPGA to implement a 64-bit double-precision matrix multiplier.

Researchers have tried to decrease the deep networks' computational workload. Using fixed-point data format [25] is an attractive way. The authors of [26] propose the idea that many connections in deep CNN (especially in full connection layers) are unnecessary that they can be pruned to reduce the computational workload and memory space. The pruned connection owns a weight of zero. Thus, the weight matrix may become a sparse matrix. The authors of [27, 28] accelerate the computation of the sparse matrix using GPGPU.

6. CONCLUSION

In this paper, we have proposed a CNN accelerator architecture that can handle a broad spectrum of network structures efficiently. We focus on accelerating the convolutional layers and full connection layers that constitute most of the CNN's computational workload. We adopt a matrix multiplier-based accelerator architecture. Effort is made to optimize the data arrangement, memory access, and the workload balancing among the accelerator PEs. To make it easy to use, we extend the Caffe framework to FPGA. A prototype system is built, a performance speed-up of 3.54× is achieved compared with an Intel Xeon X5675 CPU, and the energy efficiency is 4.7× better than an Nvidia K20 GPGPU.

ACKNOWLEDGEMENTS

The authors gratefully acknowledge supports from the National Nature Science Foundation of China under NSFC 61272145, National High Technology Research and Development Program of China (863 Program) under No. 2012AA012706, Research Fund for the Doctoral Program of Higher Education of China under SRFDP No. 20124307130004; the guidance from Prof. Xing Cai is gratefully acknowledged in connection with the revision of this paper, during the first author's research stay at Simula Research Laboratory and University of Oslo in Norway, between September and November 2015.

DOI: 10.1002/cpe

REFERENCES

1. Krizhevsky A, Sutskever I, Hinton GE. Imagenet classification with deep convolutional neural networks. *Advances in Neural Information Processing Systems*, Curran Associates, Inc., Lake Tahoe, Nevada, USA, 2012; 1097–1105.
2. Hinton G, Deng L, Yu D, Dahl GE, Mohamed Ar, Jaitly N, Senior A, Vanhoucke V, Nguyen P, Sainath TN, Kingsbury B. Deep neural networks for acoustic modeling in speech recognition: the shared views of four research groups. *Signal Processing Magazine, IEEE* 2012; **29**(6):82–97.
3. Girshick R, Donahue J, Darrell T, Malik J. Rich feature hierarchies for accurate object detection and semantic segmentation. *2014 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*: IEEE, Columbus, OH, USA, 2014; 580–587.
4. Gokhale V, Jin J, Dundar A, Martini B, Culurciello E. A 240 G-ops/s mobile coprocessor for deep neural networks. *2014 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*: IEEE, Columbus, OH, USA, 2014; 696–701.
5. Ovtcharov K, Ruwase O, Kim JY, Fowers J, Strauss K, Chung ES. Accelerating deep convolutional neural networks using specialized hardware. *Microsoft Research Whitepaper*: Microsoft Research, 2015.
6. Wan L, Li K, Liu J, Li K. GPU implementation of a parallel two-list algorithm for the subset-sum problem. *Concurrency Computation Practice Experience* 2015; **27**(1):119–145.
7. Tang G, Yang W, Li K, Ye Y, Xiao G, Li K. An iteration-based hybrid parallel algorithm for tridiagonal systems of equations on multi-core architectures. *Concurrency Computation Practice Experience* 2015; **27**(17):5076–5095.
8. NVIDIA. *TESLA K20 GPU accelerator board specification*, July 2013.
9. Farabet C, Poulet C, Han JY, LeCun Y. CNP: an FPGA-based processor for convolutional networks. *FPL 2009. International Conference on Field Programmable Logic and Applications, 2009*: IEEE, Prague, Czech Republic, 2009; 32–37.
10. Zhang C, Li P, Sun G, Guan Y, Xiao B, Cong J. Optimizing FPGA-based accelerator design for deep convolutional neural networks. *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*: ACM, Monterey, CA, USA, 2015; 161–170.
11. Cadambi S, Majumdar A, Becchi M, Chakradhar S, Graf HP. A programmable parallel accelerator for learning and classification. *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*: ACM, New York, NY, USA, 2010; 273–284.
12. Gupta S, Agrawal A, Gopalakrishnan K, Narayanan P. Deep learning with limited numerical precision. *arXiv preprint* 2015. arXiv:1502.02551.
13. Jia Y, Shelhamer E, Donahue J, Karayev S, Long J, Girshick R, Guadarrama S, Darrell T. Caffe: convolutional architecture for fast feature embedding. *Proceedings of the ACM International Conference on Multimedia*: ACM, Orlando, Florida, USA, 2014; 675–678.
14. Collobert R, Kavukcuoglu K, Farabet C. Torch7: a matlab-like environment for machine learning. *BigLearn, NIPS Workshop*, Granada, Spain, 2011.
15. Farabet C. Towards real-time image understanding with convolutional networks. *Ph.D. Thesis*, Université Paris-Est, 2013.
16. Dou Y, Vassiliadis S, Kuzmanov GK, Gaydadjiev GN. 64-bit floating-point FPGA matrix multiplication. *Proceedings of the 2005 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*: ACM, New York, NY, USA, 2005; 86–95.
17. Gorman Mel. *Understanding the Linux Virtual Memory Manager*. Prentice Hall Upper Saddle River: New Jersey, USA, 2004.
18. Xiao T, Qiao Y, Shen J, Yang Q, Wen M. Unified virtual memory support for deep CNN accelerator on SoC FPGA. In *Algorithms and Architectures for Parallel Processing*. Springer: Zhangjiajie, China, 2015; 64–76.
19. Farabet C, Martini B, Corda B, Akselrod P, Culurciello E, LeCun Y. Neuflow: a runtime reconfigurable dataflow processor for vision. *2011 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*: IEEE, Colorado Springs, CO, USA, 2011; 109–116.
20. Yi D, Lei Z, Liao S, Li SZ. Learning face representation from scratch. *arXiv preprint* 2014:arXiv:1411.7923.
21. Chakradhar S, Sankaradas M, Jakkula V, Cadambi S. A dynamically configurable coprocessor for convolutional neural networks. *ACM SIGARCH Computer Architecture News* 2010; **38**(3):247–257.
22. Peemen M, Setio AAA, Mesman B, Corporaal H. Memory-centric accelerator design for convolutional neural networks. *2013 IEEE 31st International Conference on Computer Design (ICCD)*: IEEE, 2013; 13–19.
23. Chen T, Du Z, Sun N, Wang J, Wu C, Chen Y, Temam O. Diannao: a small-footprint high-throughput accelerator for ubiquitous machine-learning. *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*: ACM, 2014; 269–284.
24. Chetlur S, Woolley C, Vandermersch P, Cohen J, Tran J, Catanzaro B, Shelhamer E. CuDNN: Efficient primitives for deep learning. *arXiv preprint* 2014:arXiv:1410.0759.
25. Jiang J, Hu R, Mikel L, Dou Y. Accuracy evaluation of deep belief networks with fixed-point arithmetic. *Computer Modelling & New Technologies* 2014; **18**(6):7–14.
26. Han S, Pool J, Tran J, Dally WJ. Learning both weights and connections for efficient neural networks. *arXiv preprint* 2015:arXiv:1506.02626.
27. Li K, Yang W, Li K. Performance analysis and optimization for SPMV on GPU using probabilistic modeling. *IEEE Transactions on Parallel and Distributed Systems* 2015; **26**(1):196–205.
28. Yang W, Li K, Mo Z, Li K. Performance optimization using partitioned SPMV on GPUs and multicore CPUs. *IEEE Transactions on Computers* 2015; **64**(9):2623–2636.

DOI: 10.1002/cpe