A Parallel Loading Based Accelerator for Convolution Neural Network

Kong Anmin and Zhao Bin

Abstract—**Convolutional neural networks (CNNs) are widely used in modern Artificial Intelligence (AI) systems. Compared with other classical methods, CNNs have superior performance in image classification, speech recognition and object detection. However the computational load of CNNs is very heavy and a large amount of data movement are expected. An efficient way of data movement is critical for both performance and power efficiency for an accelerator design. In this paper we propose a novel CNN accelerator architecture with unique parallel loading scheme and smart memory addressing solution. Our solution is 30% faster than others on Alexnet. Our proposal can achieve high efficiency for FC layer without using image batching. This will make our solution very suitable for edge applications.**

Index Terms—**Convolutional neural networks (CNNs), deep learning, energy-efficient accelerators.**

I. Introduction

Convolutional neural networks (CNNs) [1], also known as deep learning [2] are widely used in modern Artificial Intelligence (AI) systems. Before the appearance of CNNs, most of the pattern recognition algorithms are composed of a hand-crafted feature extraction module followed by a classifier. However, there are some draw backs in this kind of hand-crafted feature extraction algorithms. First, the performance of the algorithm highly depends on the hand-crafted feature. The performance of the algorithm can be very poor if a wrong feature is chosen. And, for many tasks, people don't even know how to select those features. Second, the features we selected for on task can't be applied in another task. And this will limit the application of the algorithm. To solve the above mentioned problem, CNNs [1] are proposed.

In CNNs, all the features are automatic learned from the raw data by many convolution layers. There are no human interaction in between the raw data and the final outputs. CNNs have a much deeper architectures and thus it has much more capacity to learn very complexity features than before. Thus, CNNs have the superior performance in image recognition [3]-[6], speech recognition [7], [8] and computer vision [9]-[12].

However, state-of-the-art CNNs normally have hundreds of megabytes of weights and it requires billions of operations in an inference. The data movement in an inference can be very huge. In order to process CNNs in real-time, both high parallelism in computing and high efficient data movement are required.

In this paper, we propose a novel CNN accelerator structure. It can support parallel data loading and thus can minimize the IO time. The proposed structure can fully support full connection layer efficiently. A smart addressed memory access method is also proposed. The proposed structure can support different CNN networks by re-configure the network. In this paper we use Alexnet [3] to benchmark the performance of the proposed architecture.

Manuscript received September 19, 2019; revised June 24.

The authors are with the Institute of Microelectronic, Agency for Science, Technology and Research (A*STAR), 2 Fusionopolis Way, #08-02 Innovis Tower A, Singapore (e-mail: kong_anmin@ime.a-star.edu.sg, zhaobin@ime.a-star.edu.sg).

II. Background

CNNs can achieve state-of-the-art performance in image classification, speech recognition and object detection [13]. It is constructed by stacking multiple convolutional layers (CONV) for feature extraction and full connection (FC) layers for classification.

For a CNN, convolution layers read the feature maps generated by previous layers and output new feature maps for next layer. Pooling layer is inserted between some convolution layers to reduce the size of feature map as well as the computational load of the following layers. Pooling layer can work against overfitting [14] and it can also provide a form of translation invariance [15], [16]. FC layers are used to do classification. The last FC layer outputs the probability of each category that the input image might belongs to.

For convolution layer, the primary computation is the high dimensional convolutions. Assume f_j^{in} stands for j-th input feature map, f_i^{out} stands for i-th output feature map. $g_{i,j}$ stands for the convolution kernel corresponding to the j-th input channel and i-th output channel. The convolution layer can be expressed as [15]

$$f_i^{out} = \sum_{i=1}^{n_{in}} f_j^{out} \otimes g_{i,j} + b_i \, , (1 \leq i \leq n_{out}) \qquad (1)$$

where n_{in} and n_{out} are the length of input channel and output channel, respectively. $\otimes$ is a matrix convolution operation. After the convolution, activation functions such as rectified linear unit (ReLU) [17] are applied to introduce nonlinearity.

For full connection layer, assume the input signal f^{in} is a $n_{in} \times 1$ vector the output signal f^{out} is a $n_{out} \times 1$ vector. n_{in} and n_{out} are the length of the input and output feature vector, respectively. Full connection layer can be expressed as [15]

$$f^{out} = W f^{in} + b \qquad (2)$$

where W is a $n_{out} \times n_{in}$ weight matrix, b is a $n_{out} \times 1$ vector which stands for the bias.

III. System Architecture

A. Overview

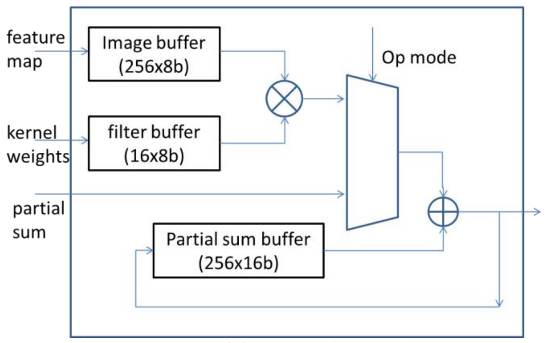


Fig. 1. Structure of the Processing Engine (PE).

Fig. 1 shows the structure of our process engine (PE). It contains one 8 bit multiplier, one adder, 256-byte image buffer, 16 byte kernel buffer and 256-byte partial sum buffer.

The convolution operation can be divided into 3 different stages. The first stage is the data loading stage. In this stage, image data and weight data are loaded into image buffer and filter buffer, respectively. Note that, in order to reduce the total loading time, loading of image data and filter weights can be carried out at the same time. The second stage is the convolution stage. In this stage, data from image buffer and data from filter buffer are multiplied and are accumulated, the results are stored in partial sum buffer. The last stage is the partial-sum accumulation stage. In this stage, partial sum at the same PE column will be accumulated accordingly from the bottom PE to the top PE module. And the results are stored in the partial sum buffer at the top PE module.

For example, suppose we want to do a 5×5 convolution, we need to use 5 PE modules and these 5 modules are stacked in column direction to form a 5×1 PE column. Each PE will be used to do the convolution of one image row with one of the 5×5 filter row. And the partial sums are accumulated accordingly to get the final results. Thus, we need to arrange the 5 PEs in column direction and the final results are accumulated and stored in the top PE. Note that, the partial sum accumulation stage and the next data loading stage can be carried out at the same time to increase the throughput of the system.

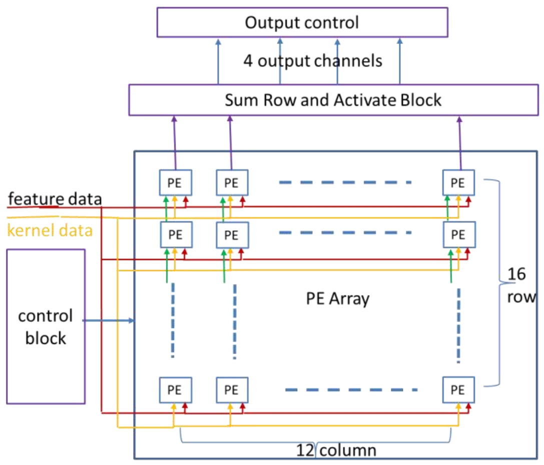


Fig. 2. Structure of the PE array.

The architecture of the PE array is shown in Fig. 2. Our PE array has 16 PE rows and 12 PE columns. Totally, there are 192 PEs in the array.

At the partial sum accumulation stage, partial sum from the bottom row of the PE array will be send out and accumulate with the partial sum from the second bottom PE row. This process will repeat till all the partial sums are accumulated at the top most PE row. The output partial sum from the top most PE row is sent to "sum row and activate block". The function of this block is to divide the 12 partial sums into 4 groups (3 in each group) and sum together. Activation functions such as rectified linear unit (ReLU) [17] are applied in this module to introduce nonlinearity. Output of the sum row and activate block is sent to max-pooling module or a data buffer for next round processing.

In our architecture, each PE row share one image data bus and one filter weight bus. Totally, there are 16 parallel image data buses and 16 filter weight buses. Independent image and filter weight bus allow us to load them simultaneously. Note that, there are 12 independent PE column enable signal. Image data/filter weight can be loaded in parallel to one (only one PE column is enabled) or to multiple PE columns (multiple PE columns are enabled) at the same time. By using this parallel loading feature, the data loading time can be greatly reduced.

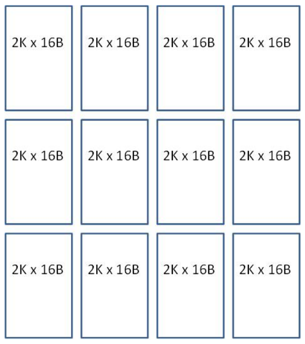


Fig. 3. Structure of the memory block.

To support this parallel loading feature, the data buffer sub-block is designed to be 16×8b width and 2K depth with byte-write support. Byte-write is the key points of our smart memory access solution. The maximum number of output channels is 4 in our design. It can process 4 output kernels at the same time. Fig.3 shows the memory structure. We partition the memory into 3×4 sub-blocks, 4 blocks in a group. We also propose a unique way of memory access control to support parallel loading and writing patterns required. The design overhead is negligible compared with normal memory access method. This will be discussed in detail later.

B. Data Operation

2D Kernel are convoluted with 2D input feature map to form partial-sum, then partial-sums from different input feature maps are summed together to get the final output feature map.

In our design, all the multiplications are calculated within PE, while the accumulations are divided into three stages. In the first stage, one row of kernel are convoluted with one row (or multiple rows) of input image. In the second stage, partial-sums from the same PE column are accumulated accordingly from the bottom PE row to the top PE row and the final partial sums are stored in the top PE row. In the third stage, dedicated adder is used to accumulate the partial-sum from different PE columns. In our design, every 3 PE columns are summed together and thus we can get the final output of 4 different output channels.

By using this architecture, we can maximize the reuse of the image and kernel data with minimum hardware resource. A dedicated max-pooling block is also designed to support different requirements of different algorithms. 2D max-pooling operation is divided into 2 one-dimension max-pooling operation. By doing so, the hardware design cost of the max-pooling can be greatly reduced and the data can flow smoothly without additional control. This will reduce the complexity of memory access, and we can remove the input/output buffer which is required by other solution for data reformatting [18], [15].

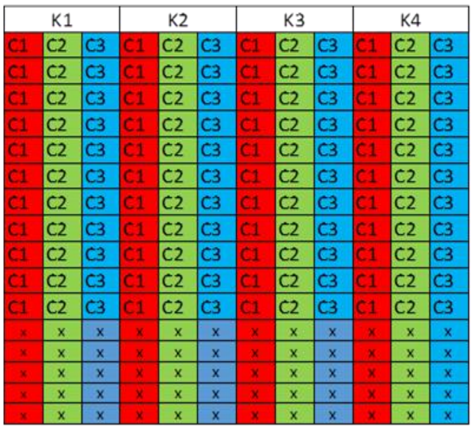
C. Convolution Layer Implementation

1) 11×11 convolution

Table I shows the shape parameter of Alexnet [3]. Here, H is the size of input feature map, R is the size of kernel, E is the size of output feature map. C is the number of input feature maps. F is the number of output feature maps. G is the number of image groups.

TABLE I: ALEXNET SHAPE PARAMETERS

Alexnet Shape Parameters						
Layer	H	R	E	C	F	G(group)
CONV1	227	11	55	3	96	1
CONV2	31	5	27	48	128	2
CONV3	15	3	13	256	192	2
CONV4	15	3	13	192	192	2
CONV5	15	3	13	192	128	2



K1			K2			K3			K4		
C1	C2	C3	C1	C2	C3	C1	C2	C3	C1	C2	C3
C1	C2	C3	C1	C2	C3	C1	C2	C3	C1	C2	C3
C1	C2	C3	C1	C2	C3	C1	C2	C3	C1	C2	C3
C1	C2	C3	C1	C2	C3	C1	C2	C3	C1	C2	C3
C1	C2	C3	C1	C2	C3	C1	C2	C3	C1	C2	C3
C1	C2	C3	C1	C2	C3	C1	C2	C3	C1	C2	C3
C1	C2	C3	C1	C2	C3	C1	C2	C3	C1	C2	C3
C1	C2	C3	C1	C2	C3	C1	C2	C3	C1	C2	C3
C1	C2	C3	C1	C2	C3	C1	C2	C3	C1	C2	C3
C1	C2	C3	C1	C2	C3	C1	C2	C3	C1	C2	C3
C1	C2	C3	C1	C2	C3	C1	C2	C3	C1	C2	C3
x	x	x	x	x	x	x	x	x	x	x	x
x	x	x	x	x	x	x	x	x	x	x	x
x	x	x	x	x	x	x	x	x	x	x	x
x	x	x	x	x	x	x	x	x	x	x	x
x	x	x	x	x	x	x	x	x	x	x	x

Fig. 4. Data mapping for 11×11 convolution.

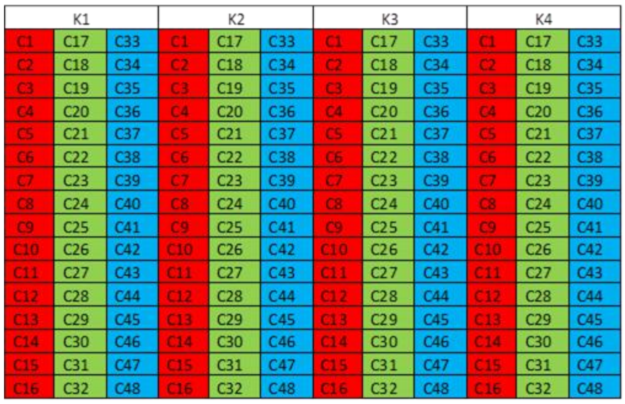
Fig. 4 shows the data mapping of CONV1 using our PE array. There are 3 input channels in CONV1 (C1, C2 and C3, as shown in the figure), and the kernel size is 11×11. One PE is used to calculate the convolution of one image row and one kernel row. Thus, 11 PEs in one PE column can be used to complete the 11×11 convolution.

Three PE columns are used to handle 3 input channels accordingly (C1, C2 and C3, respectively). The same input data will be duplicated 3 times to fully use 12 column of PE array as shown in Fig. 4. Totally, we can process 4 output kernels (K1, K2, K3 and K4 as shown in Fig. 4) at the same time. The adder within the "sum row and activate block" is configured to sum 3 columns into one output to get the final results. The output data is shifted out sequentially, and row-based max-pooling can be executed directly.

Data sharing at this layer can be described as follows: first, enable PE column 1, 4, 7, 9 at the same time; second, load the same image data to these 4 PE columns at the same time (since these 4 columns share the same data). The image data from channel C2 and C3 can be loaded in the same way. The total data loading time can be greatly reduced by using the proposed data sharing method.

2) 5×5 convolution

There are 48 input channels in CONV2 and the kernel size is 5×5. The data mapping for CONV2 is shown in Fig. 5. Each PE column can process 16 input channels (c1-c16, as shown in Fig. 5). We need 3 PE columns to process 48 input channels (c1-c48). Note that, at each round we can process only one row (totally, there are 5 rows to be processed) of 48 input channels from 4 different output kernels (K1, K2, K3 and K4 as shown in the figure). So we need another 4 rounds to complete the full convolution for 4 different output channels.



K1			K2			K3			K4		
C1	C17	C33	C1	C17	C33	C1	C17	C33	C1	C17	C33
C2	C18	C34	C2	C18	C34	C2	C18	C34	C2	C18	C34
C3	C19	C35	C3	C19	C35	C3	C19	C35	C3	C19	C35
C4	C20	C36	C4	C20	C36	C4	C20	C36	C4	C20	C36
C5	C21	C37	C5	C21	C37	C5	C21	C37	C5	C21	C37
C6	C22	C38	C6	C22	C38	C6	C22	C38	C6	C22	C38
C7	C23	C39	C7	C23	C39	C7	C23	C39	C7	C23	C39
C8	C24	C40	C8	C24	C40	C8	C24	C40	C8	C24	C40
C9	C25	C41	C9	C25	C41	C9	C25	C41	C9	C25	C41
C10	C26	C42	C10	C26	C42	C10	C26	C42	C10	C26	C42
C11	C27	C43	C11	C27	C43	C11	C27	C43	C11	C27	C43
C12	C28	C44	C12	C28	C44	C12	C28	C44	C12	C28	C44
C13	C29	C45	C13	C29	C45	C13	C29	C45	C13	C29	C45
C14	C30	C46	C14	C30	C46	C14	C30	C46	C14	C30	C46
C15	C31	C47	C15	C31	C47	C15	C31	C47	C15	C31	C47
C16	C32	C48	C16	C32	C48	C16	C32	C48	C16	C32	C48

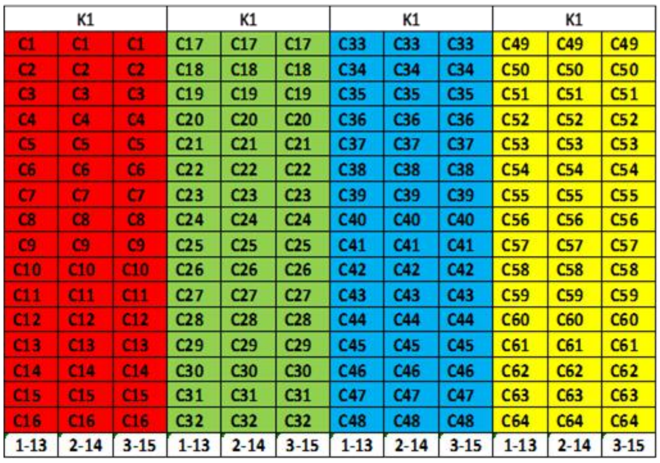
Fig. 5. Data mapping for 5×5 convolution.

Our PE is designed to support the convolution of multiple image rows with the same kernel weight. For the case of CONV2, we can load 8 rows of input feature map into image buffer at the first round (since the length of the feature is only 27 in CONV2). 8 rows of partial-sum will be calculated and stored in the partial-sum buffer. At the second round, feature map data from row 1 will be replaced by data from row 9 (because row 2 – row 9 will be used for the second round convolution), filter weights from the second filter row will also be loaded. And the corresponding partial-sum will be accumulated accordingly. Noted that in this round, the feature map loading time is only one row (that is, load row 9 to replace row 1), feature map data of row2-row8 are reused and we don't need to load them at this round. In the third round, row 3 will be replaced by row 10. In the fourth round, row 4 will be replaced by row 11 and in the fifth round, row 5 will be replace by row 12. Thus, the total loading time for the first 5 round is only 12 image rows. By using this method, we can reduce data loading time to 1/8, compared to tradition method. Note that data in the same color (column 1, 4, 7 and 10, etc…) can be shared and they can be loaded at the same time as we proposed for CONV1.

3) 3×3 convolution

In CONV3-5, kernel size is 3×3. We use 3 PE columns to complete the 3×3 convolution for one input feature map.

Thus, 16×3 PE array can process 16 input channels (c1-c16, as shown in Fig. 6). The whole 16×12 PE array can process 64 input feature maps at one round.



K1			K1			K1			K1		
C1	C1	C1	C17	C17	C17	C33	C33	C33	C49	C49	C49
C2	C2	C2	C18	C18	C18	C34	C34	C34	C50	C50	C50
C3	C3	C3	C19	C19	C19	C35	C35	C35	C51	C51	C51
C4	C4	C4	C20	C20	C20	C36	C36	C36	C52	C52	C52
C5	C5	C5	C21	C21	C21	C37	C37	C37	C53	C53	C53
C6	C6	C6	C22	C22	C22	C38	C38	C38	C54	C54	C54
C7	C7	C7	C23	C23	C23	C39	C39	C39	C55	C55	C55
C8	C8	C8	C24	C24	C24	C40	C40	C40	C56	C56	C56
C9	C9	C9	C25	C25	C25	C41	C41	C41	C57	C57	C57
C10	C10	C10	C26	C26	C26	C42	C42	C42	C58	C58	C58
C11	C11	C11	C27	C27	C27	C43	C43	C43	C59	C59	C59
C12	C12	C12	C28	C28	C28	C44	C44	C44	C60	C60	C60
C13	C13	C13	C29	C29	C29	C45	C45	C45	C61	C61	C61
C14	C14	C14	C30	C30	C30	C46	C46	C46	C62	C62	C62
C15	C15	C15	C31	C31	C31	C47	C47	C47	C63	C63	C63
C16	C16	C16	C32	C32	C32	C48	C48	C48	C64	C64	C64
1-13	2-14	3-15	1-13	2-14	3-15	1-13	2-14	3-15	1-13	2-14	3-15

Fig. 6. Mapping of 3×3 convolution.

The data mapping is shown in Fig. 6. We can load the whole feature map into one PE (since the feature map size is only 13×13, which can be placed in the image buffer directly).

Note that feature map row1-row13, row2-row14 and row3-row15 are required by PE column 1, 2 and 3 respectively (as shown in Fig. 6), which corresponding to filter row 1, filter row 2 and filter row 3, respectively. It can be observed that, data from row3-row13 can be shared between these 3 PE columns.

The data sharing process can be described as follows: first, enable PE column 1 and load data row 1 to PE column1; second, enable PE column 1 and PE column 2, load data row2 to PE column 1 and 2; third, enable PE column 1, 2 and 3, load data row3-row 13 to PE column 1, 2 and 3; fourth, enable PE column 2 and 3, load data row 14 to PE column 2 and 3; fifth, enable PE column 3, load data row 15 to PE column 3. The data loading time can be greatly reduced by using this data sharing method.

D. Full Connection Layers Implementation

For FC layer, the kernel size and feature map size are the same. But number of kernel is huge and there is no kernel reuse. Most neural-network accelerator architectures [18] [15] are designed for kernel reuse and have to use image batching method to improve the efficiency of their design. We find that if we swap the feature map data and kernel data, feature map data can be reused for FC layer, we can still achieve high-efficiency for FC layer without using image batching. We add a MUX in the image bus and kernel bus. It will allow us to control which data is loaded into image buffer and kernel buffer accordingly. By doing this, we can easily reduce the number of data loading to 1/16 in FC layers.

The operation of the FC1 can be described as follows: for each PE, we load 16 feature map data into kernel buffer, load 16 set of kernel coefficients into image buffer. Each set contains the corresponding filter weights for that 16 feature map. We can process 16×12×16 (because we have 12 columns) out of 6×6×256 feature maps in one round. Thus, after 3 rounds of convolution, we can get the results of 16 output channels. Totally we need 4096/16×3=768 rounds to finish FC1. Note that, the PE utilization is 100% in FC1 and over 99% for other FC layers as shown in Table II.

E. Smart Addressed Memory Access Method

In order to support parallel loading, the results of 16 output channels must be stored in one memory sub-block so that they can be read out in parallel. Thus, the memory sub-block is designed to be 16×8b width and 2K depth with byte-write support. The data writing process with and without max-pooling are different. We will describe this process in detail in this section.

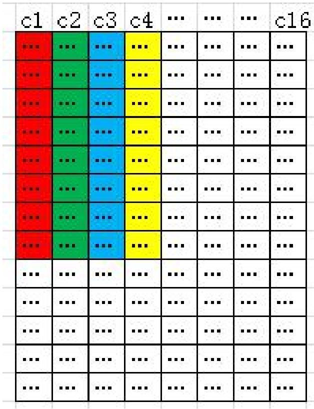


Fig. 7. Data arrangement for normal write.

Assume there is no max-pooling, at each round of convolution we can get the results of 4 output channels. The data of these 4 channels are written in parallel to the left most 4 columns in one memory sub-block (as shown in Fig. 7). Thus, after 4 rounds, we can have all 16 output channels in the memory sub-block and all the data are arranged in the suitable format for parallel loading.

However, for the case of max-pooling, the writing operation is different. In our design, max-pooling is divided in two stages. The first stage is to do max-pooling in row direction. Here we use CONV1 as an example. At the first round, input image size is 11×227, this image will be loaded into image buffer and convolute with an 11×11 filter. The output feature size is a 1×55 vector. This data is fed into max-pooling block directly and the output is a 1×27 vector. It is written in byte format (from left to right in row direction) to the memory sub-block (as shown in Fig. 8(b)). Note that, this is the output of one row of the feature map. At each round, we can process 4 output channels. These 4 output channels will be written to 4 different sub-blocks simultaneously. After first stage max-pooling, the feature map size changed from 55×55 to 55×27. Fig. 8(c) shows the data storage format after the first stage max-pooling.

The second stage is to do max-pooling in column direction, feature map size changed from 55×27 to 27×27. Note that, in order to load data in parallel at CONV2, the first row of channel 1- 16 needs to be stored in parallel in one memory sub-block (for example, data from channel 1 stored at column 1, data from channel 2 stored at column 2, etc…). The process of the second stage max-pooling can be described as follows: read 4 columns of the 55×16 feature map data from the memory sub-block (the actual data size been read is 55×4, only 4 columns are read. Because the max-pooling block can only support 4 inputs at a time). After the second stage max pooling, the data size changed to 27×4, which corresponding to 4 rows of an output channel. These data are written back to 4 different memory sub-block simultaneously, as shown in Fig. 8(d). Note that these are the first 4 rows of channel one. Data from channel 2-16 can be written in the same way. By using this memory access method, the output data can be written in the format required by parallel loading. No further data transformation is required.

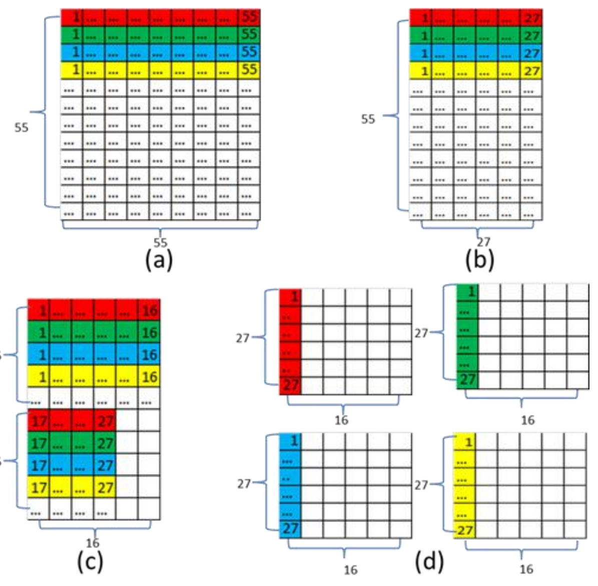


Fig. 8. Data arrangement for max-pooling.

The above write operation can be easily met by swapping the sequence of write-address. For example, we can shift 4 LSB of write address to the left, so these 4 bits can be used to control the byte-write, it allow us to write data horizontally as shown in Fig. 8(c). By using this method, the read/write address generation can be simplified to a +1 operation and a bit-swapping control of the write address.

F. Performance Estimation

Performance of the proposed accelerator (benchmarked to Alexnet) can be found in Table II. Here we assume the system clock is 200Mhz. Data loading time, kernel loading time and calculation time is estimated per rounded based.

TABLE II: PERFORMANCE ESTIMATION

layer	data load	filter load	Psum load	Conv process	total cycle	time(ms)	PE Utilization
conv1	672	132	0	605	1009800	5.049	68.75
pooling1	55		0		42240	0.2112	
conv2	744/93	60	0	1080	1409856	7.04928	100
pooling2	27		0		27648	0.13824	
conv3	780	36	169	507	841302	4.20651	100
conv4	780	36	169	507	630892	3.15446	100
conv5	780	36	169	507	422380	2.1119	100
pooling3	13		0		6656	0.03328	
FC1	192	3072	0	256	2703360	13.5168	100
FC2	192	3072	0	256	1203840	6.0192	99.8
FC3	192	3072	0	256	295680	1.4784	99.2

In CONV1, image data is reused. And we will load all the 96 different kernels first to process the same input image data. The detailed data processing can be described as follows: first, load 1 image row to image buffer. Note that there are 3 different input channels, so the total image loading time is 3×224=672; second, load filter weights to filter buffer. The total filter loading time is 11×12=132 (because we need to load filter weights to12 PE columns). The convolution time is 11×55=605. Note that at each round, we can only process 4 output kernels. And we will process 96 output kernels first for the same image. Thus, the total time to get one row of the final results of 96 kernels is 672+(132+605)×96/4. And we need to process 55 rows. Thus, the total clock cycle for CONV1 can be calculated as 55×(672+(132+605)×96/4).

In CONV2, we use the data sharing method mentioned in section C to reduce the data loading time. The detailed operation can be described as follows: In round 1, we load 8 feature map rows and the total cycle is 31×8×3=744. However in next round, we only need to load row 9 (to replace row 2) and the data loading time is just 31×3=93 cycles). Compare with [18], our CONV2 takes only 7.05 ms (70% of [18]). The reason is that our PE array is fully used in CONV2 and the data loading time is minimized by using parallel loading.

In CONV3-CONV5, we reuse the input feature map as proposed in section C. The cost is an extra partial sum loading (13×13=169 clock cycles per round). Note that our PE array is also fully used in CONV2-CONV5 and FC1, that is the reason why it has a better performance compare with other method [18].

TABLE III: PERFORMANCE COMPARISON

Alexnet Layers	CONV1	CONV2	CONV3	CONV4	CONV5	FC1	FC2	FC3
Y.H.Chen[1]	5.23	10.48	5.9	4.6	2.63	NA	NA	NA
our method	5.05	7.05	4.21	3.15	2.11	13.52	6.02	1.48
speed up	1.04x	1.49x	1.4x	1.46x	1.25x			

IV. CONCLUSION

In this paper we propose a novel CNN accelerator structure. Compared with other methods, the proposed structure has the following advantages: 1, data is load into PE in parallel; 2, the proposed structure can fully support FC layer efficiently; Other methods [18] [15] use image batching to improve the efficiency of FC layer which can cause large latency and is not suitable for EDGE application. 3, we propose a smart memory access solution. It enables us to reformat the data sequence directly. And we don't need input/output data buffer, which are required by other proposals. 4, the proposed method can outperform [18], and the hardware resource requirement is minimized.

CONFLICT OF INTEREST

The authors declare no conflict of interest.

AUTHOR CONTRIBUTIONS

Kong anmin and zhao bin concuted the research together; They also analyzed the data together; kong anmin wrote the paper, zhao bin help to verify and revise the paper; all authors had approved the final version.

REFERENCES

[1] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*, MIT Press, 2016.

[2] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[3] 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.

[4] C. Farabet, C. Couprie, L. Najman, and Y. LeCun, "Learning hierarchical features for scene labeling," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 35, pp. 1915–1929, 2013.

[5] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *Proc. International Conference on Learning Representations*, 2014.

[6] C. Szegedy, W Liu, Y. Jia *et al.*, *Going Deeper with Convolutions*, arXiv preprint arXiv: 1409.4842, 2014.

[7] T. Mikolov, A. Deoras, D. Povey, L. Burget, and J. Cernocky, "Strategies for training large scale neural network language models," in *Proc. Automatic Speech Recognition and Understanding*, pp. 196–201, 2011.

[8] T. Sainath, A.-R. Mohamed, B. Kingsbury, and B. Ramabhadran, "Deep convolutional neural networks for LVCSR," in *Proc. Acoustics, Speech and Signal Processing*, 2013, pp. 8614–8618.

[9] P. F. Felzenszwalb, R. B. Girshick, D. Mcallester, and D. Ramanan, "Object detection with discriminatively trained part-based models," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 32, no. 9, p. 1627, 2010.

[10] P. Viola and M. J. Jones, "Robust real-time object detection," *Int. J. of Comput. Vision*, vol. 57, no. 2, p. 87, 2001.

[11] J. T. Qiu, J. Wang, and S. Yao, "Going deeper with embedded FPGA platform for convolutional neural network," in *Proc. the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 26-35.

[12] C. Chen, A. Seff, A. L. Kornhauser, and J. Xiao, "Deepdriving: Learning affordance for direct perception in autonomous driving," in *Proc. ICCV*, 2015.

[13] B. Wu, F. Iandola, P. H. Jin, and K. Keutzer, "Squeezedet: Unified, small, low power fully convolutional neural networks for real-time object detection for autonomous driving," ArXiv preprint arXiv: 1612.01051, 2016.

[14] V. Athanasios, D. Nikolaos, D. Anastasios, P. Eftychios, "Deep learning for computer vision: A brief review," *Computational Intelligence and Neuroscience*, 2018, pp. 1-13, 2018.

[15] F. B. Tu, S. Y. Yin, and P. Ouyang, "Deep convolutional neural network architecture with reconfigurable computation patterns," *IEEE Trans. On VLSI systems*, vol. 25, issue 8, Aug. 2017.

[16] K. Fukushima, "Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position," *Biological Cybernetics*, vol. 36, no. 4, pp. 193–202, 1980.

[17] V. Nair and G. E. Hinton, "Rectified linear units improve restricted Boltzmann mchines," in *Proc. 27th Int. Conf. Mach. Learn. (ICML)*, 2010, pp. 807-814.

[18] Y. H. Chen, T. Krishna, and J. S. Emer, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE Jouranl of Solid-State Circuits*, vol. 52, issue 1, Jan. 2017.

Kong Anmin received B.S. degree in electronic engineering from the University of Electronic Science and Technology of china, Chengdu, China, in 1999 and the Ph.D degree in electric and electronic engineering from Nanyang Technological University, Singapore in 2007.

Currently he works in Institute of Macroelectronics, A*Star, Singapore as a research scientist. His research interests include deep learning, design and implementation of deep learning accelerator.

Zhao Bin received the B.S. degree from Peking University in 1990, majoring in microelectronics. He then received M.S. and M.Eng from Chinese Academy of Sciences and NUS in 1993 and 1998 respectively.

He has worked in Trident Technology Pte Ltd (Beijing Branch), Myson Technology Pte Ltd (Beijing Branch) before he moved to Singapore. From 1998 to 2000, he worked at Philips Electronics Pte Ltd. Since July 2000, he joined IME, A*Star and now as a senior research engineer. He has involved in many digital IC design projects, such as WLAN Baseband, ONFIG ONU unit, RFID tag/reader, EHSII, ZigBee Baseband, NVM memory etc. His current interesting is on the high-efficient AI accelerator architecture design and IC implementation.