C-Brain: A Deep Learning Accelerator that Tames the Diversity of CNNs through Adaptive Data-level Parallelization

Lili Song, Ying Wang, Yinhe Han, Xin Zhao, Bosheng Liu, Xiaowei Li
State Key Laboratory of Computer Architecture
Institute of Computing Technology, Chinese Academy of Sciences, Beijing, P.R. China
{songlili, wangying2009, yinhes, zhaoxin, liubosheng, lxw}@ict.ac.cn

ABSTRACT

Convolutional neural networks (CNN) accelerators have been proposed as an efficient hardware solution for deep learning based applications, which are known to be both compute-and-memory intensive. Although the most advanced CNN accelerators can deliver high computational throughput, the performance is highly unstable. Once changed to accommodate a new network with different parameters like layers and kernel size, the fixed hardware structure, may no longer well match the data flows. Consequently, the accelerator will fail to deliver high performance due to the underutilization of either logic resource or memory bandwidth. To overcome this problem, we proposed a novel deep learning accelerator, which offers multiple types of data-level parallelism: inter-kernel, intra-kernel and hybrid. Our design can adaptively switch among the three types of parallelism and the corresponding data tiling schemes to dynamically match different networks or even different layers of a single network. No matter how we change the hardware configurations or network types, the proposed network mapping strategy ensures the optimal performance and energy-efficiency. Compared with previous state-of-the-art NN accelerators, it is possible to achieve a speedup of 4.0x-8.3x for some layers of the well-known large scale CNNs. For the whole phase of network forward-propagation, our design achieves 28.04% PE energy saving, 90.3% on-chip memory energy saving on average.

1. INTRODUCTION

Deep Convolutional Neural Network algorithms are gaining popularity in machine learning and making breakthroughs in many fields, such as image recognition[1], automatic speech recognition[2] and video recognition[3]. Deep learning also begin to migrate into smartphones, wearable devices, solving real world problems in robot vision, surveillance and driver-less cars[4]. Unfortunately, such deep learning algorithms are highly time-consuming and require large amount of computing resources. Due to the computational requirements of deep learning, various NN accelerators have been proposed recently to make it inexpensive and ubiquitous for embedded or even cyber-physical applications.

From the aspect of hardware platform, most CNN acceleration solutions are based on GPGPU[5], FPGA[6], ASICs and application-specific neural processor[7]. The GPGPU solution is too cumbersome to be used in low-power platforms, embedded applications or even cost-sensitive data centers. Compared to GPU-based system, FPGA and ASIC are more attractive approaches to map the NN to hardware, because they possess advantages of high performance and energy efficiency. However, they are not flexible enough to handle a myriad of complex NN models from different areas. Worse still, application-specific NN accelerators fully expand the topology of a NN model, which is a power and area disaster for large scale Deep NNs.

Therefore, a general purpose neural processor (NP) like[8] and [9] is thought as a promising solution to offer both flexibility and efficiency. Such a NP has many good features. First, it reuses the limited hardware resources in a time multiplexing way to increase hardware and power utility. Second, it relies on multi-aspect data tiling methods to exploit data locality and relieve the pressure to on-chip memory. Last of all, NP are often designed to support a wide range of NN models. With all these good merits, state-of-the-art NPs also face some common design weaknesses, and still have a huge space for both performance and power optimization. One of the key issues is the severe performance variation to different network parameters of different models. We found in experiments that an important reason for this drawback is that most of current NPs rely on a comparatively rigid method to exploit the data-level parallelism and fixed data tiling policy in NNs. They are pursuing data-level parallelism as it does in conventional vector processor architectures. However, NPs for deep learning accelerators are basically intended to address the huge space of NN algorithms in the most energy-efficient way. More specifically, rigid vector machine structure often fails to deal with the complexity of data flows or fully exploit the multi-aspect parallelism in diverse NNs, leading to the under-utilization of hardware resources and precious memory bandwidth.

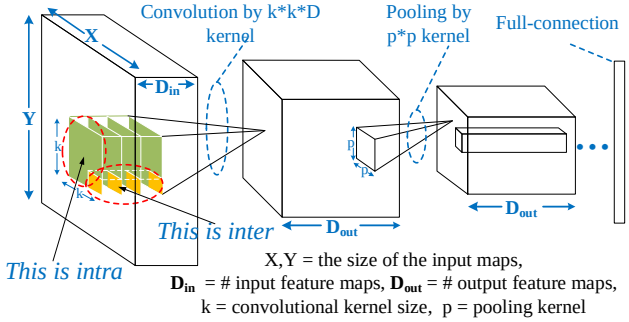

Figure 1. A typical CNN containing multiple layers

Taking a typical CNN illustrated in Fig. 1 for example, the forward propagation of a CNN includes repetitive layers of kernel-level operations, like convolution and pooling, which are the critical tasks to accelerate for NPs. Generally, there are two major types of data-level parallelism to exploit in such kernel operations: inter-kernel and intra-kernel parallelization. Exploiting them with NP will induce quite different memory access behaviors (details will be introduced in section 4).

According to our observation, different networks or even different layers of the same network have distinct parameters, so sticking to one type of parallelism and data mapping policy cannot fit all network topologies. In this paper, we proposed a kernel-

DAC '16, June 05-09, 2016, Austin, TX, USA
 ISBN 978-1-4503-4236-0/16/06…$15.00
DOI: http://dx.doi.org/10.1145/2897937.2897995

partitioning scheme accompanied by a new data tiling method to eliminate the dependencies between adjacent convolutional windows, which increase the parallelism remarkably. What's more, we investigated an adaptive data mapping scheme for large NNs to fundamentally reduce memory-traffic demand of the accelerator, ensuring the optimal performance and energy-efficiency under various types of networks and hardware configurations.

In summary, we make the contributions as follows:

-We proposed a kernel partitioning scheme that pursues both intra-kernel and inter-kernel parallelism to accelerate the convolutional layers in CNN as a hybrid approach. It partitions the original kernel into properly-sized tiles to eliminate the overlapping between adjacent kernel windows and better preserve data locality, resolving the problem that it is hard to accelerate the critical bottom layers in prior designs due to data thrashing.

-We proposed an adaptive data-level parallelization scheme for hardware CNN accelerator which combines inter-kernel, intra-kernel parallelism and hybrid (kernel-partitioning) according to network parameters and hardware resources. The experiments proved that this dynamic scheme can optimize performance and minimize energy consuming simultaneously.

-We designed and implemented the deep learning accelerator that support adaptive data tiling and parallelization schemes. The proposal is evaluated with multiple state-of-the-art large NN architectures, e.g. Alexnet[1], GoogleNet[10], VGG[3].

2. RELATED WORK

Early CNN accelerators are focused on data-path optimization. [11] and [12] mainly utilize parallelism within feature maps and kernel, and they cannot scale to various NN types and layers. [13] pursues "inter-output" and "intra-output" parallelism, but does not use on-chip buffers for data reuse and give little concern to locality exploitation. [7] organizes data path according to the sliding window property of convolutional layers, but it also ignores the data reusing patterns of feature maps.

Another class of accelerators put enough emphasis on memory-level optimization in CNNs. [6] chooses to maximize date reuse and minimize bandwidth requirement, but the addressing and data mapping are quite complicated and rely on the reconfigurability of FPGA to process different layers. [14] also takes advantage of data locality, and balances the resources of bandwidth and FPGA computation power. However, they just give a solution for Alexnet. The design philosophy in [8], which focuses on memory bandwidth utilization, can be applied to different NN layers. However, they use the same data-level parallelism and tiling scheme for different networks, leading to the underutilization of hardware resources under some sceneries.

In contrast, our work outperforms previous approaches for the following reasons. First, we seek to provide a hybrid parallelization with a novel data partitioning policy to better preserve locality. Second, instead of keeping a fixed data-level parallelization scheme for all layers, our architecture offers multiple ways of NN parallelization. The data tiling and mapping policy is changed according to the parameters of neural layers to increase the data reusability and moves the data fetch operations off the critical path of NN accelerator. Therefore, our design can adaptively switch among different parallelization strategies and the corresponding data mapping method to dynamically match different networks or even different layers of a single network.

3. PRIMER ON DEEP CNN AND DEEP LEARNING ACCELERATOR

Deep CNN are typically organized into interleaved convolutional and pooling layers followed by a number of fully-connected layers as illustrated in Fig. 1. In this paper, we primarily discuss convolution operation, which typically makes 90% of the computational workload of a CNN[12]. Fig. 1 illustrates the basic pattern of convolution. An input cube is convolved with D_{out} groups of kernels ($D_{in}\times k\times k$) at stride s. Each kernel is shifted in a sliding-window (with an offset s) across the multiple input maps. During each shift, every weight belonging to the kernel is multiplied to the according input element in the input maps and then added-up. And then an optional pooling operation (defined by p and s_p) is used to subsample the convolved output. Fig. 2 is a typical architecture of state-of-the-art deep learning accelerators [8, 13, 14], which consists of four main components: one input data buffer, one output data buffers, one weight buffer, a computational block (neural Processing Unit, PE) and a logic Control Unit (CU). There is always a compiler, executed on host platform, that automatically translate network specification (numbers of layers, kernel size etc.) written by domain experts into a code segment, which can be mapped, scheduled and executed on the accelerator. Once the instructions are ready, the raw image data and weights of pre-trained model are injected into the external memory as the input. And then, CU reads instructions one by one, loads data and weights to on-chip buffer, and computing. The accelerator performs forward propagation layer by layer and finally output the results to the external memory.

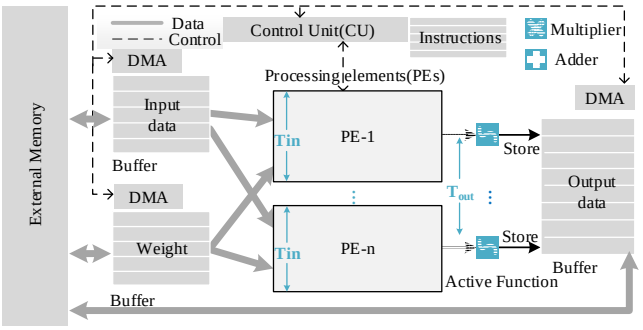

Figure 2. Architecture of a typical deep learning accelerator

Based on the deep CNNs and the general purpose deep learning accelerator architecture, we discuss two major computation acceleration policy: inter-kernel and intra-kernel parallelization, and propose a hybrid method that combines their advantages and avoid their disadvantage. In addition, there are also two kinds of data tiling schemes in coordination with the parallelization schemes, which will be stated in the following sections. To extract the best performance out of CNNs with distinct layer parameters and diverse topology, we also propose an adaptive method to adaptively switch between the three parallelization schemes for different stages of NN propagation.

4. METHODOLOGY

4.1 Analyzing the pros and cons of uniform parallelization schemes

4.1.1 Inter-kernel parallelization

Inter-kernel parallelization is to transfer n pixels in the D_{in} direction, which belong to same kernel position but different input maps (Fig. 1), to the computing unit PEs (Fig. 2). Each operation loads N <data-weight> pairs to PEs and this procedure repeats for the whole kernel window (k×k), then accumulate the result of k×k operations and send the sum to output buffer.

This approach can straightforwardly map input data from memory to PEs, but its maximum parallelism degree is restricted by the dimension of input and output maps (D_{in} and D_{out}). Take Alexnet [1] as example, the c1,c2 and c3 are different convolution

layers(D_{in}=3,48,256 respectively), and number '16' and '32' are T_{in} (the number of multiplier of one PE)in Fig. 2. When the number of input maps well matches T_{in}, the real performance is equal to the ideal performance, c2-16, c3-32, etc. Otherwise, some PE computing resources are wasted, like c2-32, especially for c1 (the number of input maps is just 3, so 13 PEs unutilized when T_{in} = 16). It can be inferred that with the T_{in} becomes wider, more and more computing resources will be wasted, leading to poor scalability. The same logic also applies to D_{out} and T_{out}.

More importantly, the inter-kernel parallelism ignores the important characteristic of convolution layers, kernel sharing, so data reuse rate is quite low. Since the concurrent data in PE belong to different input maps in depth, each operation has to reload and flush the data and weight. Thus, the buffer traffic is very heavy, leading to high power consumption. But in general, inter-kernel is easier to be implemented and its performance is relatively good for the layers with big input maps (D_{in}).

4.1.2 Intra-kernel parallelization

Different from inter-kernel parallelism, intra-kernel parallelism is to transfer one or several k×k windows in same input maps to PE. Because map size is mostly bigger than map depth (X×Y>Din, in Fig. 1), intra-kernel parallelism is more efficient compared to inter-kernel. However, due to the characteristics of convolution and diverse parameters of different layers, the kernel sizes and data folding schemes are various, which in turn leading to complicated data alignment and data mapping from memory to PE. According to our study, there are three approaches to handle the problem.

1. Data unrolling. This scheme is easy to do the mapping, but creates extremely large foot-print size due to data duplication and need to reshape data layout before move to next layer of CNN. For example, given a 28×28 map with k = 5 and s = 1, after unrolling, the data map size is 24 ×24×25. Thus the on chip buffer size and memory traffic will be enlarged for almost (k/s) × (k/s) times. Data duplicates for T times as given by Equation 1:

$$T=\frac{\left((\boldsymbol{X}-\boldsymbol{k})/\boldsymbol{s}+\boldsymbol{1}\right)\times\left((\boldsymbol{Y}-\boldsymbol{k})/\boldsymbol{s}+\boldsymbol{1}\right)\times\boldsymbol{k}\times\boldsymbol{k}}{\boldsymbol{X}\times\boldsymbol{Y}} \qquad \textbf{Equation 1}$$

Where X, Y are the sizes of input maps, s is stride, and k is kernel size. Fig. 3 shows the first five Conv layers in Alexnet and GoogleNet, the unrolled data size increases to 9x~18.9x of the raw input. In addition, it's difficult to reshape data from raw data to the unrolled one in hardware, so it sometimes relies on a host possessor to do that at considerable overhead. So this method is always used in software system[18], not in hardware accelerator.

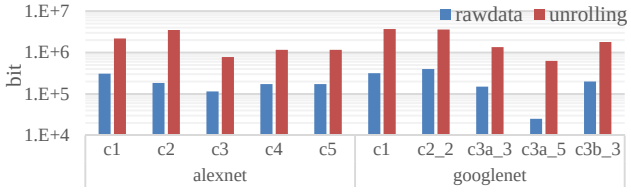

Figure 3. Data unrolling scheme

2. Sliding window. Another approach is to use sliding window, which has quite high requirements for data alignment. It is efficient only when k is equal to s, so that no data overlapping exists between two adjacent kernel windows. We just need to put data within the same kernel window to the buffer in sequence. Data pixels in the same map are continuously loaded together from buffer to PE. In most cases, k is bigger than s. There are overlapping of two adjacent kernel windows both X direction and Y direction, as show in Fig. 4(a). No matter how to arrange the raw data to buffer(X direction or Y direction), after shifting the window, the data in same window will distributed across buffer lines (or rows). As a result, the memory access intensity increases because requests have to be issued for several times to load data from buffer to PE. When kernel size is much bigger or smaller than T_{in} in PE, the memory access pattern will become more complicated. In that case, the performance and power cost will be prohibitively high for an efficient accelerator.

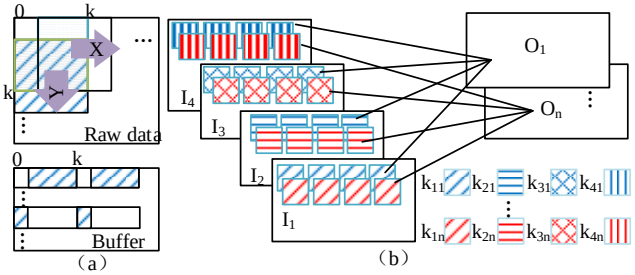

Figure 4. Sliding window and kernel sharing

3. 2D-PE. Additional, there is another method that realizes the intra-kernel level parallelization, which is a different structure of 2D mesh PE similar to systolic array [11, 15]. This design exhibits a very high data reusability and is very effective when dealing with specific network topology in vision processing. However, this highly-effective 2D-PE design will encounter performance degradation or underutilization issue when it encounters networks with varied size of kernels and stride.

However, intra-kernel parallelism has a big advantage in terms of energy reduction. It is based on the fact that the concurrent data in PE belong to the same input maps and share same kernel across the whole layers, so each operation just need to reload either data or weight in buffer, not both. Fig. 4(b) gives an example of 4 input maps and n output maps with 4*n groups of weights. For weight k_{11} is shared in input map I_1, so we can keep k_{11} in PE and load different data windows in I_1, until the whole input map is traversed completely, then the fixed weight k_{11} will never be re-accessed. We also can keep one data window of I_1 in PE and sequence load k_{11} to k_{1n}, then the fixed data window will never be re-accessed. Thus, the buffer traffic is reduced dramatically compared to inter-kernel in most cases. As the analysis of [8], buffer traffic is the largest part of energy consumption, so it is remarkable in whole system energy reduction.

4.2 The proposed parallelization scheme

Based on above analysis, both inter and intra-kernel parallelization schemes have advantages and disadvantages. In term of performance, the former is usually better except for the critical bottom layers ([1, 3, 10, 16]), which are obviously more suitable to be accelerated by intra-kernel parallelization. What we have to do is to deal with the data alignment issue of intra-kernel parallelization and find an easy way to map it onto hardware.

Therefore, in this section we propose a new parallelization scheme to effectively accelerate the critical bottom layers and the latter top layers by combining the advantages of both inter- and intra-kernel parallelization schemes. In brief, we design the method "**kernel-partitioning**" to make intra-kernel scheme easy to map data onto hardware with low memory and buffer bandwidth consumption. Also, to accelerate the top layers with more number of maps, we improve the inter-kernel scheme to utilize the locality of data and weight, thus cost much less power.

In the end, we also process the adaptiveness to effectively accelerate the different networks or different layers. No matter how we change the hardware configurations or network topologies, the proposed network mapping strategy ensures good data reusability and easy alignment in memory and buffer, contributing to the optimal performance and energy-efficiency.

4.2.1 *Kernel partitioning*

The challenge to implement the intra-kernel parallelization is the following two aspects. First, kernel window size is much bigger than PE width (T_{in}). Second, the stride is smaller than the kernel size. To resolve the problems once for all, we partition the kernel window depending on the stride as shown in Equation2:

$$g = ceil(k / s), \quad k_s = s \qquad \textbf{Equation 2}$$

Where g is the number of pieces that a big kernel is partitioned into, and k_s is the new kernel size after partitioning.

Fig. 5 shows the details to partition a kernel. Taking Alexnet Conv1 for example, Fig. 5(a) shows the raw data. Since the length and height of the data are not dividable by k_s, '0's are padded at the boundary. Fig. 5(b) is the mapping result from the small kernel windows ($k_s \times k_s$, represented by $d_{x,y}$) to the on-chip buffer in sequence. Fig. 5(c) shows the layout of the corresponding weights, one more line of '0' is also padded. The weights are partitioned into g×g pieces, with size = $k_s \times k_s$. Each small window of weight is represented by $w_{i/9}$. Specific steps to do the partitioning are described in Algorithm 1.

Algorithm 1

Input:

k:kernel,s:stride,k_s:kernel after partition, g:the groups of partion,T_{in}:the number multiplier in a PE, $size_x$,$size_y$:the size of output maps, G=g×g

1:For i=1:G
2: load $w_{i/G}$ to PE
3: FOR j_x = i%g:(i%g+$size_x$),r_x = 1:$size_x$
4: FOR j_y=(i/g+1):(i/g+$size_y$),ry=1:$size_y$
5: Mapping d(j_x,j_y) to PE
6: Calcuate
7: IF i=1, THEN store the result to buffer as a pixel located at (r_x,r_y) of output map $r_{i/G}$(in Fig. 5d)
8: ELSE reload pixel(r_x,r_y)of $r_{(i-1)/G}$ from buffer, add the MAC result to it, then store the sum as a pixel(r_x,r_y) of $r_{i/G}$
9:END END END END

For example, the original big kernel (11×11) is partitioned into 9 small sub-kernels (4×4) in Fig. 5(c), so the code within the first outer loop of Algorithm 1 accomplishes 1/9 computing tasks of the original big kernel. The data (Fig. 5a) multiplied to the first sub-piece of weights (Fig. 5c) is starting from $d_{1,1}$ to $d_{55,55}$, from left to right and then top to bottom. Second one is starting from $d_{1,2}$ to $d_{56,56}$. The same calculation method applied to the following 7 sub-kernels. Thus the last piece of weights is multiplied to the data from $d_{3,3}$ to $d_{57,57}$. Ultimately, there will be 9 output maps as shown in Fig. 5(d), with map size to be 55×55. So the final result is to add the 9 maps together, which is exactly the same with the original computing method with that of big kernels.

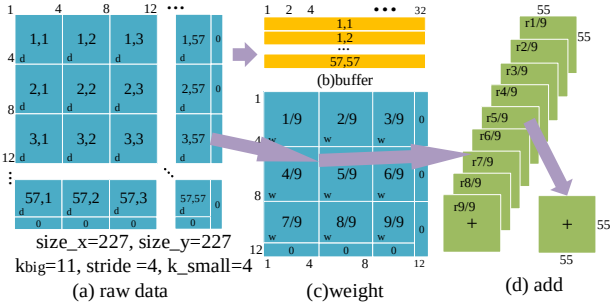

Figure 5. Kernel partitioning

The mapping scheme is same to intra-kernel parallelism (section 4.1.2), but the basic unit is a small kernel window ($k_s \times k_s$). When T_{in} is bigger than the size of small kernel window ($k_s \times k_s$), we map multiple small windows to PE in one operation.

4.2.2 *Improvement for inter-kernel parallelism*

As is mentioned at the beginning of section 4.2, in terms of performance, inter-layer parallelism is already good enough for subsequent top layers of NN. However, considering the energy consumption, inter-kernel parallelism ignores the important characteristic of convolution layers-kernel sharing within the maps, so data reuse rate is quite low. In this section, we proposed an improvement for the mapping scheme of inter-kernel parallelism to increase the data reuse rate.

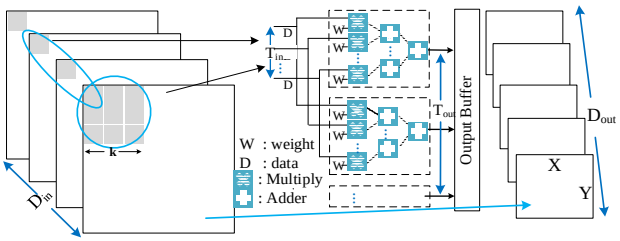

Figure 6. Mapping scheme of kernel partitioning

As shown in Fig. 6, there are D_{in} input maps and D_{out} output maps with $D_{in} \times D_{out}$ groups of weight, and kernel size is k. Prior designs go through the whole kernel window at first, which means the kernel window will not shift to right or downward until it accomplished a complete pixels of the output map (need $k \times k \times D_{in}$ times of multiplication). They have to reload both data and weight repetitively on each mul-operation. In our design, to better reuse both data or weight for less memory access, each time we move to the same pixel in the next output map or the next pixel in the same output map to calculate the 1/(k×k) partial sum instead of the complete sum. In this case, the output buffer is used to store the partial sums, and it requires additional "add-and-store" operations to accumulate the partial sums to obtain the final result. However, this method has better data or weight reusability and so reduces buffer loading at a cost of additional buffer store operations. For instance in Fig 6, accumulating the partial sums induces $X \times Y \times D_{out} \times k \times k$ more times of store operations, but it saves almost $X \times Y \times D_{out} \times k \times k \times D_{in}/T_{in}$ times of load operations. Since D_{in} is always much bigger than T_{in} in top layers, this method dramatically decreases buffer bandwidth occupancy. Besides, store is thought off the critical path of computation.

Compared with inter-kernel parallelization scheme, we minimized memory or buffer access traffic, at the same time, maximized the use ratio of logic resources.

4.2.3 *Providing Self-adaptiveness*

Because the structures of different neuron layers in a network have entirely distinct kernel parameters, they must be processed with different parallelization schemes instead of a fixed policy to achieve the optimal performance. According to our statistics, the arrangement of neural layers in deep CNNs follows a rule that the bottom layers always have big kernels and small number of input maps. As the network goes deeper with feature abstraction, the feature size becomes smaller and smaller, the kernel size shrinks and the number of input maps increases. Therefore, the three parallelization schemes are complementary to each other in many aspects as demonstrated in Table 1, which gives us an opportunity to hybrid them together in one round of NN propagation.

Table 1. Parallelization scheme comparison

scheme	Suited layer characteristic	Advantages
Inter	Large #input maps and small kernel	Implement easily
Intra	Kernel = stride	Less memory traffic
partition	Big kernel or small #input maps	Both of above

Algorithm 2 shows the algorithm rule of how to select the properly scheme to accelerate NN computing.

Algorithm 2

Given a NN layer:
1: IF k=s and k≠1, THEN select intra-kernel parallelism
2: ELSE-IF D_{in} <T_{in},THEN select kernel-partition
3: ELSE Select inter-kernel parallelism
4:IF(paralellism scheme of nextlayer is inter-kernel),
store in inter-order(Din,X,Y in Fig. 1)
5:ELSE Stroe in intra-order(X,Y,Din in Fig. 1)
Move to next NN layer

To orchestrate different parallelization schemes in a whole network, we proposed a smart data tiling and mapping scheme to cooperate with it, which optimizes the data layout in memory so that the weight and data of one neuron layer is mapped and aligned according to the needed parallelization scheme of this layer. Our design requires no special hardware like rotatable buffers[6] or data layout transformation unit [8].

5. EVALUATION

5.1 Experimental setup

To evaluate our proposal, we implemented a Verilog based CNN accelerator. We synthesized the implementation using Synopsys Design Compiler (DC, under TSMC 45nm library), then verified the design using Synopsys VCS. Additionally, we also designed a compiler that generates the macro instruction flow for the accelerator.

We implement different parallelism schemes into the accelerator for comparison. In this paper, we just consider the inference operation (net forwarding). Since the network models are pre-trained, the accuracies of NNs are fixed under different parallelism schemes, and we just care about speed and energy consumption of NN forward-propagation. We recorded the cycles of simulation to quantify the performance and evaluated the power consumption based on the synthesized results of DC.

Benchmark. We selected several recent popular large NN structures, Alexnet[1], GoogleNet[10], VGG[3] and Nin[16]. The first two are the champions of ILSVRC[17] 2012, 2014 respectively, which is one of the most popular object detection and image classification challenges over the world. The characteristics of these network architectures are shown in Table 2. Note that the first row in the table is the detailed parameters of conv1, and the data are #input maps, k, s, #output maps respectively. Second row is the number of Conv-layers of NN. Third row is the kernel size used in the whole NN.

Table 2. Benchmark

Network	**Alexnet**	**google net**	VGG	NiN
Conv1 detail	3,11,4,96	3,7,2,64	3,3,1,64	3,11,4,96
#conv layers	5	57	16	12
Kernel types	11,5,3	7,5,3,1	3	11,5,3,1

Experimental settings. To fully evaluate the performance of our accelerator design, we compared all of the following schemes based on our accelerator, including inter-kernel, intra-kernel, and kernel-partition parallelization with adaptive mapping scheme. Table 3 gives the accelerator parameter. Please note that the data width of the neural processing element(PE) is 16-bit fixed-point, which is validated to be good enough with reference of [8]. We set the ideal performance to be the upper bound performance, which assumes that all the computing components are 100% utilized at running time and data alignment is also ideal without wasting any buffer space and bandwidth.

Table 3. Accelerator parameters

name	bandwidth	size	opertation	cycle
PE	16-16,32-32	16bit	mulitplication	1
InOut-buf	16,32	2M Byte	add	1
Weight-buf	256,1024	1M Byte	load	1
Bias_buf	16,32	4K Byte	store	1

5.2 Performance

1. In order to compare the performance of kernel-partition scheme with the other two, we ran 3 groups of experiments on layer Conv1 (Conv1's performance accounts for a large portion of the whole network, because the input map size is the biggest), defined as *inter*, *intra* and *partition* in Fig. 7. Also, we set PE width(T_{in},T_{out}) to be '16-16' and '32-32' respectively.'16-16' stands for the computation engine with 16 inputs from input feature maps, 16 inputs from weights, thus the number of multipliers is 256, and the number of adder tree is 16(each with 16 adders).'32-32' is the same. *inter* are the scheme proposed in [8]. *intra* is similar to [6] except for the data reshaping method. *partition* is the kernel partition scheme we proposed in Section 4.2.1.

The left part in Fig. 7 is the experimental results when PE width is 16-16, and the right part is under 32-32. For each situation, we compare the three schemes across different NNs, including Alexnet, GoogleNet, VGG and Nin from left to right. The results show that *intra* and *partition* schemes are much better than *inter*, which almost reach the upper bound performance. This is because the number of input maps is 3 in Conv1, which is very small, and the *inter* scheme waste lots of computing resources, which is not the case in the other two strategies. Moreover, based on the analysis in Section 4.1, for *intra* scheme, sliding window is too difficult to be implemented in hardware due to the big kernel size and small stride(k=11,7,3,11;s=4,2,1,4) of Conv1, so we implemented the unrolling scheme in this paper. Since the extra memory traffic of unrolling scheme, *intra* is also slower than *partition*. To be specific, average of 4 NNs, *partition* outperforms *inter* and *intra* by a 5.8x and 2.1x speed-ups respectively.

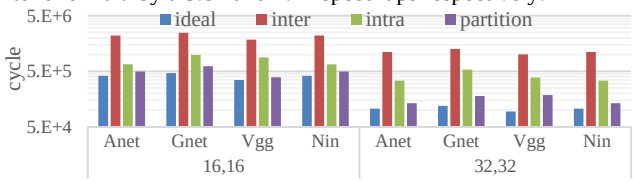

Figure 7. Comparison of execution time of layer Conv-1

2. Regarding the adaptive scheme proposed in Section 4.2.3, we ran 5 groups of experiments, including *inter*, *intra*, *partition* and *adpa-1* and *adpa-2*. The first 3 schemes are the same with the previous experiment and the only difference is that we run the whole network this time using the same scheme across all the layers of the NN. We added our adaptive scheme, which adaptively change the parallelization schemes between layers, to do the comparison. The difference of *adpa-1* and *adap-2* is that the inter-kernel parallelism in adap-1 is the original one, and adap-2 is the one improved in section 4.2.2. It is shown in Fig. 8 that the adaptive scheme outperforms the others significantly. In particular, *adpa* outperforms the most commonly used inter by 1.83x in Alexnet. As average of the 4 NNs, the speedup is 1.43x. It should be noted that there are two reasons why the speed up of adap is not so remarkable for VGG. First, the size of VGG is very large, and the biggest layer need 8M buffer, so we have to exchange data frequently between on-chip buffer and off-chip memory which is very time consuming. Second, all the layers of VGG use almost the same parameter (D_{in}≥64, k=3, s=1), the space for adaptiveness is rather marginal. Additionally, *partition* is not so good in whole round of NN propagation as previous experiment. It is because that *partition* is good for the bottom layers like Conv1, but not suitable for the top layers which have large number of input maps and very small kernel. So the *adap* is

preferable to the whole NN. Also, *adpa-1* and *adpa-2* are the same on performance, and their difference are in energy consumption, which will be presented later.

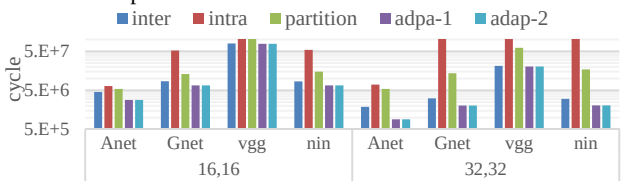

Figure 8. Performance comparison

3. To give a fair evaluation, we also compared our adaptive scheme with [14], and the details are illustrated in Fig. 9. Their method is denoted as *Zhang-7-64* in Fig. 9, where 7-64 is T_{in}-T_{out}. For fare comparison, we down-scale our design to 100MHz as is used in [16]. Since our design configuration is different with [14], we change the parameter 7-64 ,which is the optimal one proposed in [14], to 16-28, when *adap* scheme uses the same computing resources with [14]. It can be seen from Fig. 9 that our scheme *adpa-16-28* outperforms [14] with 2.22x and 1.20x speedup on Conv1 and the whole network respectively. '28' is not the optimal parameter for our design, so we add another 2 configurations in the experiments. For *adap-16-24*, the number of multipliers is 14% less than [14]; for *adap-16-32*, the number of multipliers is 14% more than [14]. For whole network, the speedups are 1.06x, 1.45x respectively. More importantly, [14] is customized for Alexnet and is thought as not competitive as ours for other NNs.

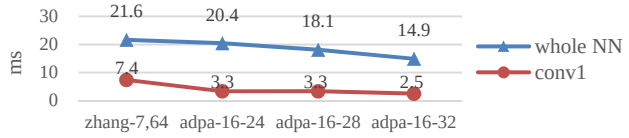

Figure 9.Performance compare with[14]

4. The performance comparison with CPU (Intel Xeon 2.20GHz) is shown in Table 4. The software implementations are written in C++ based on Caffe[18]. Our accelerator operates at 1GHz clock. Overall, our implementation *adap-16-16* and *adap-32-32* achieve up to 139.35x and 468.67x speedup (avg. of the 4 NNs) over software implementations respectively.

Table 4 Performance comparied to CPU(ms)

	CPU	adap-16-16	speedup	adap-32-32	speedup
Anet	376.50	2.83	133.02x	0.91	414.58x
Gnet	1418.8	6.69	212.11x	2.04	696.88x
Vgg	10071.71	77.51	129.94 x	20.41	493.44x
Nin	553.43	6.72	82.35 x	2.05	269.77x

5.3 Energy consumption

Table 5 gives the PE energy reduction of different schemes. *adpa-1* and *adpa-2* saves respectively 28.04% and 27.96% more energy than *inter* on average. *adap-2*'s reduction is slightly smaller than *adap-1*, because a group of adders and other combinatorial logic are added to support "add-and-store" operation in *adpa*-2. However, *adap-2* remarkably reduces the buffer access intensity, which is the major source of energy consumption in NN accelerator(also illustrated in[8]).

Table 5 PEs Energy reduction (%)

	inter(base)	intra	partition	adap-1	adap-2
Alexnet	0.00	32.85	40.23	47.77	47.71
Googlenet	0.00	9.66	22.77	31.48	31.40
VGG	0.00	-44.72	-8.61	3.00	2.89

In Fig. 10, we compare the buffer access count of the evaluated schemes for whole NNs. *adap-2* achieves 90.13% memory traffic reduction on average compared to *adap-1*. In theory, the bandwidth consumption rate of *intra* and *adap-2* should be close with each other. But in practice, since there are many redundant data dues to the data alignment problem in some layers of *intra*, *adap-2* is much better than *intra*, achieving up to 73.7% buffer traffic reduction on average.

Additionally, because of the additional "add-and-store" operations in latter top layers of VGG, partition have more buffer accesses than others, but *adpa-2* saves 93.8% and 77.6% buffer traffic on average compare to *inter* and *intra* respectively. Although the performance speedup of *adap-2* for VGG is not so conspicuous, the energy reduction is tremendous.

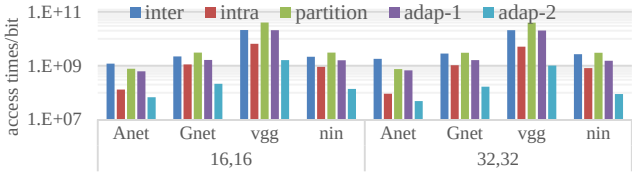

Figure 10. Buffer traffic comparison

6. Conclusion

In this paper, we proposed a general purpose deep learning accelerator which offers multiple types of data-level parallelism: inter-kernel, intra-kernel and hybrid. The design can adaptively switch among the three types of parallelism and the corresponding data tiling schemes to dynamically match different networks or even different layers of a single network. No matter how we change the hardware configurations or network topologies, the proposed network mapping strategy ensures the optimal performance and energy-efficiency.

7. ACKNOWLEDGMENTS

This work was supported in part by National Natural Science Foundation of China under Grant No. 61432017, 61176040, 61504153 and 61221062. The corresponding authors are Yinhe Han and Ying Wang.

8. REFERENCES

[1] Krizhevsky, A., et al., ImageNet Classification with Deep Convolutional Neural Networks. *Advances in Neural Information Processing Systems*, 2012.

[2] Hinton, G., et al., Deep Neural Networks for Acoustic Modeling in Speech Recognition. *IEEE Signal Processing Magazine, 2012*.

[3] Simonyan, K., et al., Very Deep Convolutional Networks for Large-Scale Image Recognition. *Arxiv*, 2014.

[4] LeCun, Y., et al., Deep learning. *Nature*, 2015.

[5] Coates, A., et al. Deep learning with COTS HPC systems. *In Proc. of ICML*, 2013.

[6] Peemen, M., et al. Memory-centric accelerator design for convolutional neural networks. *In Computer Design (ICCD)*, 2013.

[7] Farabet, C., et al. Neuflow: A runtime reconfigurable dataflow processor for vision. *In Proc. of CVPRW*, 2011.

[8] Chen,T.,et al. Diannao:A small-footprint high-throughput accelerator for ubiquitous machine-learning. *In Proc. of ASPLOS*, 2014.

[9] Park, S., et al. 4.6 A1. 93TOPS/W scalable deep learning/inference processor with tetra-parallel MIMD architecture for big-data applications. *In Proc. of ISSCC*, 2015.

[10] Szegedy, C., et al., Going deeper with convolutions. *ArXiv*, 2014.

[11] Sankaradas, M., et al. A massively parallel coprocessor for convolutional neural networks. *In Proc. Of ASAP*, 2009.

[12] Cadambi, S., et al. A programmable parallel accelerator for learning and classification. *In Proc. of PACT*, 2010.

[13] Chakradhar, S., et al. A dynamically configurable coprocessor for convolutional neural networks. *In ACM SIGARCH Computer Architecture News*, 2010.

[14] Zhang, C., et al. Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks. *In Proc. of FPGA*, 2015.

[15] Du, Z., et al. ShiDianNao: shifting vision processing closer to the sensor. *In Proc. of ISCA*, 2015.

[16] Lin, M., et al., <Network In Network>. *In Proc.of ICLR*, 2014.

[17] Russakovsky, O., et al., Imagenet large scale visual recognition challenge. *International Journal of Computer Vision*, 2014.

[18] Jia, Y., et al. Caffe: Convolutional architecture for fast feature embedding. *ArXiv*, 2014.