2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture

Prediction based Execution on Deep Neural Networks

Mingcong Song[1], Jiechen Zhao[1], Yang Hu[2], Jiaqi Zhang[1], Tao Li[1]
[1]Department of Electrical and Computer Engineering, University of Florida, Gainesville, FL, USA
[2]Department of Electrical and Computer Engineering, The University of Texas at Dallas, Richardson, TX, USA
[1]{songmingcong, jiechen.zhao, jiaqizhang}@ufl.edu, [2]yang.hu4@utdallas.edu, [1]taoli@ece.ufl.edu

***Abstract*—Recently, deep neural network based approaches have emerged as indispensable tools in many fields, ranging from image and video recognition to natural language processing. However, the large size of such newly developed networks poses both throughput and energy challenges to the underlying processing hardware. This could be the major stumbling block to many promising applications such as self-driving cars and smart cities.**

Existing work proposes to weed zeros from input neurons to avoid unnecessary DNN computation (zero-valued operand multiplications). However, we observe that many output neurons are still ineffectual even if the zero-removal technique has been applied. These ineffectual output neurons could not pass their values to the subsequent layer, which means all the computations (including zero-valued and non-zero-valued operand multiplications) related to these output neurons are futile and wasteful. Therefore, there is an opportunity to significantly improve the performance and efficiency of DNN execution by predicting the ineffectual output neurons and thus completely avoid the futile computations by skipping over these ineffectual output neurons.

To do so, we propose a two-stage, prediction-based DNN execution model without accuracy loss. We also propose a uniform serial processing element (*USPE*), for both prediction and execution stages to improve the flexibility and minimize the area overhead. To improve the processing throughput, we further present a scale-out design for *USPE*. Evaluation results over a set of state-of-the-art DNNs show that our proposed design achieves 2.5X speedup and 1.9X energy-efficiency on average over the traditional accelerator. Moreover, by stacking with our design, we can improve *Cnvlutin* and *Stripes* by 1.9X and 2.0X on average, respectively.

Keywords-Deep Learning; Accelerator; Approximate Computing; Output Sparsity

I. Introduction

Deep learning, especially deep convolutional neural networks (CNNs) [1], has achieved great success on many fields, ranging from image and video recognition to natural language processing. This is mainly due to their ability to achieve unprecedented accuracy on many challenging machine-learning problems, such as object recognition [2] and detection [3]. The improved accuracy comes along with significantly increased computation and memory demands. For example, VGGNet [4] with 548 megabytes filter weights requires 1.5×10^{10} floating-point multiplications per image to perform object recognition. Thus, the intensive computation of such newly developed networks exerts the pressure of throughput and energy efficiency on computer architecture and system design [5], [6]. Without efficient architectural supports, it hinders pervasive intelligence deployment for many promising applications such as self-driving cars [7], smart cities [8], and AI-based IoTs [9].

Towards the goal of ubiquitous learning, researchers perceive the opportunities to tap into DNN accelerators [10]–[16]. Many accelerator designs leverage pruning based technique [11], [17] to shrink the size of original, dense networks. Although pruning can reduce the computational and memory overheads, it requires retraining the pruned-network to restore the accuracy. The training procedure is time-consuming and thus loses the flexibility. For instance, it takes three weeks to train VGGNet on four high-end NVIDIA Titan GPUs. Other works devote to reducing the computational operations via approximate computing based methods [18]–[20]. Nevertheless, their performance improvement comes at the expense of some loss in network accuracy. Therefore, it is desirable to explore techniques that can effectively reduce computation and memory overheads while avoiding retraining and accuracy loss.

A promising trend is to remove the unnecessary calculations across the neuron networks and thus saving the time and energy. An initial proposal named *Cnvlutin* [21] aims at eliminating zero-valued operand multiplications to reduce computational operations without the sacrifice of accuracy. It is inspired by the fact that Rectifier Linear Unit (ReLU) results in abundant zeros in the input feature maps [22].

However, we observe that the zero-removal technique adopted by *Cnvlutin* still not fully exploits the opportunities in DNNs. First, many output neurons (after non-linearity layers such as ReLU and Max-pooling) are still ineffectual even if the zero-removal has been applied. Since an ineffectual output neuron cannot pass its value to next layer, this means all the computations (including non-zero-valued operand multiplications) related to these output neurons are futile and wasteful. Second, *Cnvlutin* mainly focuses on convolutional (CONV) layers. However, since most of the filter weights are in fully-connected (FCN) layers, *Cnvlutin* could not reduce the memory access overhead. Third, the existence of Max-pooling (MaxP) layer in DNNs greatly reduces the amount of zeros, and this can stultify the zero-removal technique of *Cnvlutin*.

Instead of simply eliminating the ineffectual operand multiplications where one of the inputs is zero, we opt to skip over the complete computation process that relates to ineffectual output neurons. In this paper, we define ineffectual output neurons (iEON) as those output neurons that have no

2575-713X/18/$31.00 ©2018 IEEE
DOI 10.1109/ISCA.2018.00068

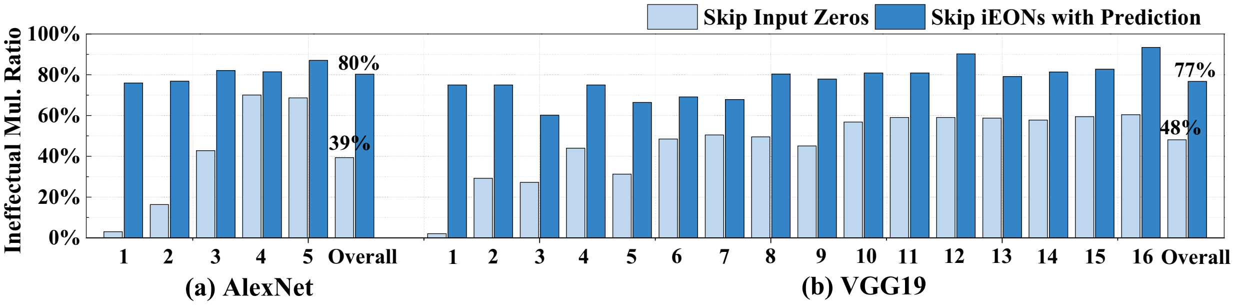


Figure 1. Comparison of Ineffectual Multiplications

influence on the subsequent layers since they are filtered out by the non-linearity layers in DNNs, such as ReLU and MaxP. For ReLU, only the output neurons with positive value can pass to the next layer. For MaxP, only the output neuron with the maximum value in a sub-region can pass to the next layer. If we can predict the iEONs in advance, we can skip over the whole computation process (including those non-zero-valued operand multiplications) relates to these iEONs and greatly improve performance.

Compared with removing ineffectual multiplications relates to zero inputs, completely skipping over iEONs manifests the following advantages: (1) it offers more benefits on performance and energy since skipping over iEONs can eliminate more multiplications in each layer as shown in Fig. 1; (2) it can also tackle the challenge of memory access since each output neuron has its own filter in FCN layers and there is no need to load the filters when their corresponding output neurons are ineffectual; (3) although MaxP is unfavorable for *Cnvlutin*, it is a good opportunity for us to explore since it produces more iEONs; and (4) it can be stacked with other ineffectual neuron removal techniques such as *Cnvlutin*. For those predicted effectual output neurons (EON), we can further reduce its zero-operand multiplications to achieve higher speedup.

To practically skip over the iEONs, we propose to transform the computing pattern of DNNs from one-time execution to a two-stage, prediction-based execution. First, the predictor predicts the EONs (Prediction Stage). Then, the executor only needs to perform computations related to EONs (Execution Stage). Further, we maximize the computational reuse between these two stages to amortize the computational overhead introduced by Prediction Stage. Specifically, the results of the predictor are not only used for prediction but also can be utilized by the executor. Thus, the executor only needs to perform the remaining calculation of EONs and obtain the final result by adding the partial result from predictor, consequently achieving near-ideal speedup and no accuracy loss. Moreover, we design a uniform serial processing element (*USPE*), for both predictor and executor to get rid of the extra area overhead.

To maximize the processing throughput, we further propose a scale-out design to process the multiple output neurons in parallel. However, the sparsity of EONs in the output feature maps challenges the scale-out design and results in the idleness of *USPE*. We leverage pre-fetch loading and out-of-order execution to make *USPE*s being fully-utilized. We also propose analytical models to demonstrate our memory design that can provide enough data for *USPE*s.

Evaluation results over a set of state-of-the-art DNNs show that our proposed design achieves an average 2.5X speedup and 1.9X energy-efficiency over the traditional accelerator. Our prediction-based design has an average 1.8X speedup over *Cnvlutin* [21] and 1.6X speedup over *Stripes* [23]. Moreover, by combining our design, we can improve *Cnvlutin* and *Stripes* by 1.9X and 2.0X on average respectively.

In summary, we make the following key contributions:

- We observe an opportunity to significantly improve the performance and efficiency of DNNs by completely bypassing the computation of iEONs brought by the non-linearity layers such as ReLU and MaxP.
- We propose a two-stage, prediction-based DNN execution model without sacrificing the accuracy. This model employs Prediction Stage to identify the EONs for CONV and FCN layers, and reuses the prediction results as intermediate results to incrementally conduct the execution on EONs.
- We present a uniform serial processing element (*USPE*), for both predictor and executor to improve the flexibility and minimize the area overhead.
- We propose the scale-out design of *USPE* to maximize the processing throughput.

The rest of this paper is organized as follows. Section II introduces the background and motivation. Section III illustrates our prediction-based microarchitecture design. Section IV describes the challenges and optimization in the scale-out design. Section V evaluates our design. Related works and conclusions are discussed in Sections VI and VII, respectively.

II. Background and Motivation

A. Non-linearity and Computation Overheads in DNNs

Deep learning techniques employ multiple neural network layers to learn levels of representation and abstraction that make sense of data such as image, sound, and text. Convolutional (CONV) and Fully connected (FCN) layers are the two most important deep learning network layers. Although CONV and FCN account for most of the computations in DNNs, as linear functions, they only form a new linear combination and do not create qualitatively new

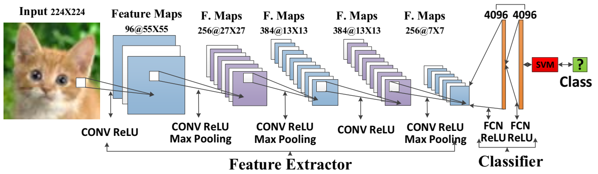


Figure 2. An Overview of CNN Model AlexNet

information. To make DNNs learn the complex functional mapping between inputs and outputs, non-linear activation functions always follow each CONV and FCN layer. Rectified linear unit (ReLU) is the most popular non-linear activation function in the state-of-the-art DNNs [24], such as ResNet [2], GoogleNet [25], and VGGNet [4]. It introduces non-linearity by only allowing positive values pass through and converting any negative input to zero. Another important technique that introduces non-linearity is pooling [26], which is a form of non-linear down-sampling. There are several non-linear functions to implement pooling among which max-pooling (MaxP) is the most common choice. It partitions the input image into a set of rectangles and, for each such sub-region, outputs the maximum pixel.

We introduce preliminaries of neural network architecture using AlexNet [1] as an example. As shown in Fig. 2, it mainly consists of five convolutional layers, two fully connected layers, seven ReLUs and three MaxPs. Convolutional (CONV) layers extract "features" by adopting a convolutional kernel on its input feature maps. The computational pattern is a convolutional operation calculated along three dimensions, as shown in (1).

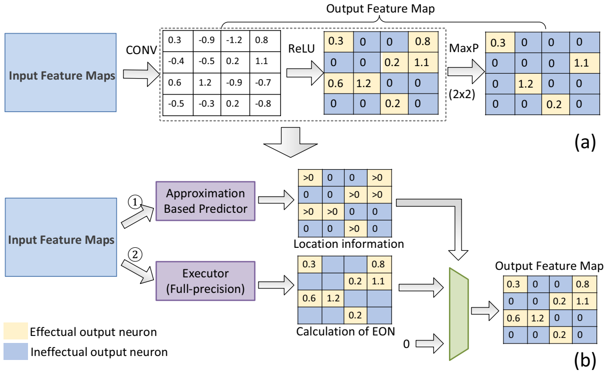
$$\underbrace{O(m,x_o,y_o)}_{\text{Output Neuron}} = \sum_{y_i=0}^{K_y-1}\sum_{x_i=0}^{K_x-1}\sum_{n=0}^{N-1} \underbrace{w^m(n,y_i,x_i)}_{\text{Filter Weight}} \times \underbrace{i(n,y_i+x_o\times S, x_i\times S)}_{\text{Input Neuron}} \quad (1),$$

where N is the number of input feature maps, S is the stride size, $K_y \times K_x$ is the filter kernel size. Note that calculating one output neuron involves $K_y \times K_x \times N$ multiply-accumulate (MAC) operations. Thus, CONV is computationally intensive. Fully-connected (FCN) layers are used to make the final inference. They take "features" in the form of a vector from a prior feature extraction layer, multiply it with a weight matrix, and output a new feature vector. The operation in FCN can also be described by (1), where K_x and K_y are equal to 1 and N is the length of each kernel. In FCN layer, each output neuron is connected to all the input neurons in the previous layer. This results in a large number of kernel weights in FCN layers and makes them memory-intensive.

B. Predicting Output Sparsity: An Opportunity for DNNs

As shown in Fig. 2, the basic network architecture combinations are CONV-ReLU, CONV-ReLU-MaxP and FCN-ReLU. We use CONV-ReLU-MaxP to exemplify the computing process in DNNs.

As shown in Fig. 3(a), there exist many negative values in the output feature maps. Since ReLU converts a negative value to zero, these negative output neurons could not pass to the next layers. After MaxP layer, even some positive output neurons could not pass to the following layers because MaxP layer only selects the maximum value in a sub-region. Thus, the EONs of ReLU are the neurons with positive values. The EONs of MaxP are the maximum neurons in a sub-region.



0.3	-0.9	-1.2	0.8
-0.4	-0.5	0.2	1.1
0.6	1.2	-0.9	-0.7
-0.5	-0.3	0.2	-0.8

0.3	0	0	0.8
0	0	0.2	1.1
0.6	1.2	0	0
0	0	0.2	0

0.3	0	0	0
0	0	0	1.1
0	1.2	0	0
0	0	0.2	0

>0	0	0	>0
0	0	>0	>0
>0	>0	0	0
0	0	>0	0

0.3			0.8
		0.2	1.1
0.6	1.2		
		0.2	

0.3	0	0	0.8
0	0	0.2	1.1
0.6	1.2	0	0
0	0	0.2	0

Figure 3. A Naive Design

If DNN accelerators can predict the EONs in advance and skip the calculation of iEONs, their performance will be greatly improved. As shown in Fig. 1, for AlexNet, the overall ineffectual multiplications related to iEONs accounts for around 80% of total multiplications, which can result in about 5.0X speedup.

C. Leveraging Predictability: A Naive Design

An intuitive approach to achieve the above-mentioned optimization is to design a dedicated predictor for the existing DNN accelerator. We show a conceptual design in Fig. 3(b). The approximation-based predictor first predicts the locations of EONs. Then the executor only needs to focus on the calculation of EONs. For the ineffectual neurons, the executor can simply assign zeros to them since they have no influence on the subsequent layers. During the prediction, we are not interested in the specific numerical value of each output neuron and only concern about whether they are positive (CONV-ReLU) or what their relative values are (CONV-ReLU-MaxP). Therefore, there is no need to perform the full-precision calculation in the prediction. Instead, we can leverage approximation-based method to design the predictor.

Challenges: Though the conceptual design can skip iEONs, there are still some design challenges that limit it to achieve ideal speedup. First, the location information of EONs obtained from the predictor could not be leveraged by the executor in Step 2. The executor still needs to perform the full-precision calculation in Fig. 3(b). Second, the integration of predictor introduces additional area overhead. Finally, the sparsity of EONs in the output feature maps challenges the traditional DNN accelerator parallelization design. As DNNs comes with intensive computational workloads, a typical solution is to perform multiple output neurons in parallel shown in Fig. 4(a). Since it simultaneously processes multiple input feature maps (IFMs) and output feature maps (OFMs), we name this architecture design as MIFM-MOFM, which is widely used in many accelerator [10], [27]–[30], such as DianNao [30] and DaDianNao [10]. As shown in Fig. 4(a), all the processing elements (PEs) in MIFM-MOFM share the same input window (I·W). However, due to this

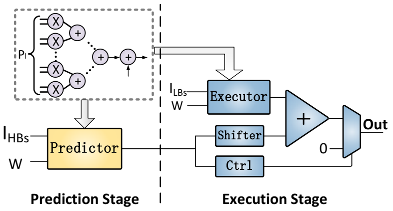


Figure 5. Two-Stage Architecture

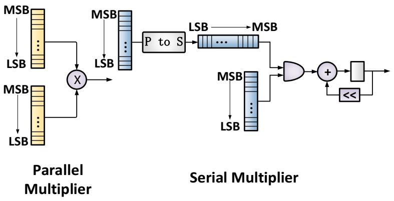


Figure 6. Parallel and Serial Multipliers

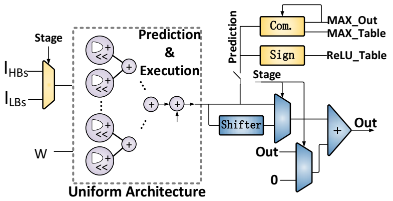


Figure 7. Uniform Serial PE (*USPE*)

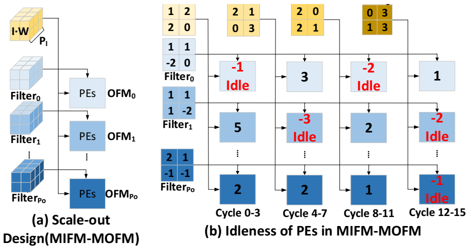


Figure 4. Idleness of PEs in Scale-out Design

sharing, MIFM-MOFM could not skip over the iEONs with negative values in Fig. 4(b), which results in the idleness of PEs and offsets the performance improvement brought by the prediction.

III. Prediction based Microarchitecture Design

As depicted in Section II, the predictability of non-linearity (ReLU and MaxP) could be leveraged to significantly reduce the unnecessary computation in linearity layers (CONV and FCN). To maximize this benefit, we need to minimize the performance offset introduced by the prediction. Towards this goal, we propose a two-stage computing pattern for the linearity layers. The computing pattern maximizes the computation reuse between the executor and the predictor. Specifically, the executor only needs to perform the remaining calculation of EONs based on the prediction results, which reduces its computational overhead. Moreover, we propose a uniform architecture *USPE*, for both predictor and executor to get rid of the extra area overhead and gain flexible configurability for computation in linearity layers.

A. Prediction Rationale (Computation Reuse)

We first introduce the rationale of prediction-based execution model for computation reduction. According to (1), the calculation of one output neuron (O) can be simplified as the sum of products of input neurons (I) and filter weights (W). As the input neurons (I) can be separated into high-order bits (I_{HBs}) and low-order bits (I_{LBs}), the calculation in (1) is equal to the sum of two parts shown in (2): the sum of products of I_{HBs} and W and the sum of products of I_{LBs} and W.

$$O = \sum W \times I = \sum W \times (I_{HBs} \ll N_{LBs} + I_{LBs})$$
$$= \underbrace{(\sum W \times I_{HBs}) \ll N_{LBs}}_{\text{Prediction Stage}} + \underbrace{\sum W \times I_{LBs}}_{\text{Execution Stage}} \qquad (2),$$

where N_{LBs} indicates the number of bits in I_{LBs} and << represents the shifting operation. Then the calculation of one output neuron can be broken down into two stages as shown in (2). Since the value of the output neuron is mainly dominated by the calculation related to high-order bits (Prediction Stage), the calculation results of Prediction Stage can be leveraged to predict the positive or negative sign of output neurons (CONV-ReLU) or their relative values (CONV-ReLU-MaxP). Therefore, as shown in Fig. 5, the predictor is responsible for the calculation of Prediction Stage and the remaining calculation (Execution Stage) is assigned to the executor. First, the predictor predicts the locations of EONs only using high-order bits of input neurons. Then, for each of these EONs, the executor continues to conduct the remaining calculation with the low-order bits of input neurons. Finally, the executor obtains the final values of EONs by adding its results with the results of Prediction Stage (reusing the results of predictor).

Recall the discussion in Section II-A, computing one output neuron involves $K_y \times K_x \times N$ multiply-accumulate (MAC) operations, which is compute-intensive. To accelerate DNN execution, as shown in Fig. 5, we equip the predictor and executor with multiple multipliers so that they can process multiple MAC operations in parallel. Specifically, we unroll its input feature maps and the parallelism granularity is P_I, which accounts for the number of multipliers.

To reduce the overhead of prediction, we should find the minimum numbers of high-order bits that is enough to perform the prediction. We conduct experiments to find the optimal choice for different layers by varying the width of high-order bits. Note that not only the input neurons could be split as high bits and low bits to provide the prediction, the high-order/low-order bit splitting also is applicable to filter weights (W). Our characterization in Fig. 8 considers both input and weight splitting methods.

We apply input splitting and weight splitting methods to CONV and FCN respectively and report the performance improvement and the number of memory access with different splitting options: weight splitting in CONV and FCN (CONV_W+FCN_W), input splitting in CONV and FCN (CONV_I+FCN_I) and a combination of two splitting methods (CONV_I+FCN_W). Comparing Option 1 (CONV_W+FCN_W) and Option 2 (CONV_I+FCN_I), we observe that input splitting method can achieve higher speedup since it requires less high-order bits, while weight splitting method has an advantage of memory access efficiency. This is because weight splitting can bypass the loading of the weights related to the iEONs in FCN layers.

TABLE I: QUANTIZING INPUT AND WEIGHT IN THE PREDICTION

Network	CONV (Input Splitting)	FCN (Weight Splitting)
Alexnet	2-2-1-1-2	5-5
VGG11	3-3-1-2-1-2-1-3	8-8
VGG13	2-3-2-3-1-2-1-2-2-3	8-8
VGG19	4-4-3-4-3-3-3-4-3-3-3-4-3-3-3-4	10-9
NiN	2-1-2-1-2-1-2-1-2-3-2-2	NA
Squ.	4-2-2-3-2-2-3-2-2-4-2-2-3-2-2-3-2-2-3-2-2-4-2-2-3	10

For 100% accuracy compared with the baseline, profiling the number of high-order bits required in the prediction

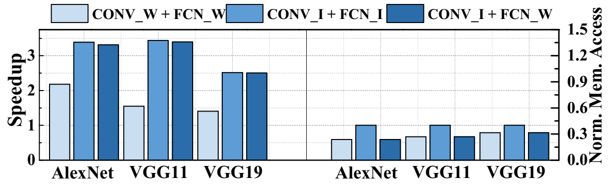


Figure 8. Splitting Weight vs Input among Different Layers

Considering CONV is compute-intensive and FCN is memory-intensive, we choose input splitting in CONV and weight splitting in FCN (Option 3). As shown in Fig. 8, this combination of two splitting methods has relatively higher speedup and fewer memory access (the best comprehensive performance). TABLE I lists the least numbers of high-order bits in different layers among various networks for prediction without accuracy loss.

Although the predictor can use a few high-order bits to perform the prediction, the two-stage design shown in Fig. 5 is far from fully leveraging this opportunity. First, the number of low-order bits in Execution Stage is greatly larger than the number of high-order bits in Prediction Stage. Worse, the input splitting varies across different layers. If we use parallel multipliers to perform the basic MAC operations, in these two stages, different networks demand their specific multiplier designs (with different bit-widths) to maximize energy efficiency, which is impractical. Furthermore, the ratio of the number of MAC operations (*nMAC*) on the two stages ($nMAC_{Execution}/nMAC_{Prediction}$) varies significantly among different layers as shown in Fig. 9. Even though we can use two specific multipliers for each stage (i.e., 4 bits for Prediction Stage and 12 bits for Execution Stage), the uneven computational ratio still complicates the two-stage pipeline design and results in pipeline bubbles. Thus, the mismatch between hardware utilization and different bit-width requirements motivates us to design a uniform architecture to support both Prediction and Execution Stages.

B. A Uniform, Serial Multiplier (Architecture Reuse)

As discussed in Section III-A, the multiplier is the basic unit in both predictor and executor. The various number of bits in these multipliers' operands motivates us to design a serial multiplier based architecture to replace the parallel multiplier as shown in Fig. 6.

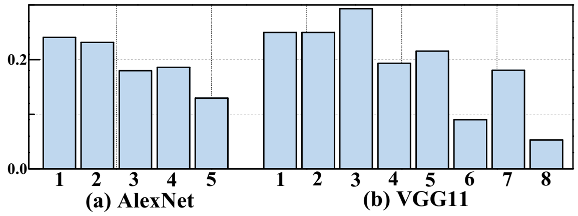


Figure 9. Uneven Computational Ratio on Two Stages

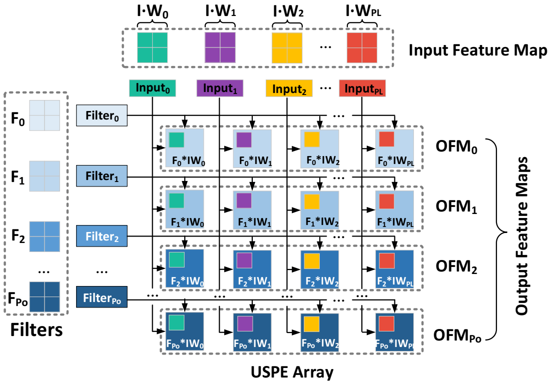


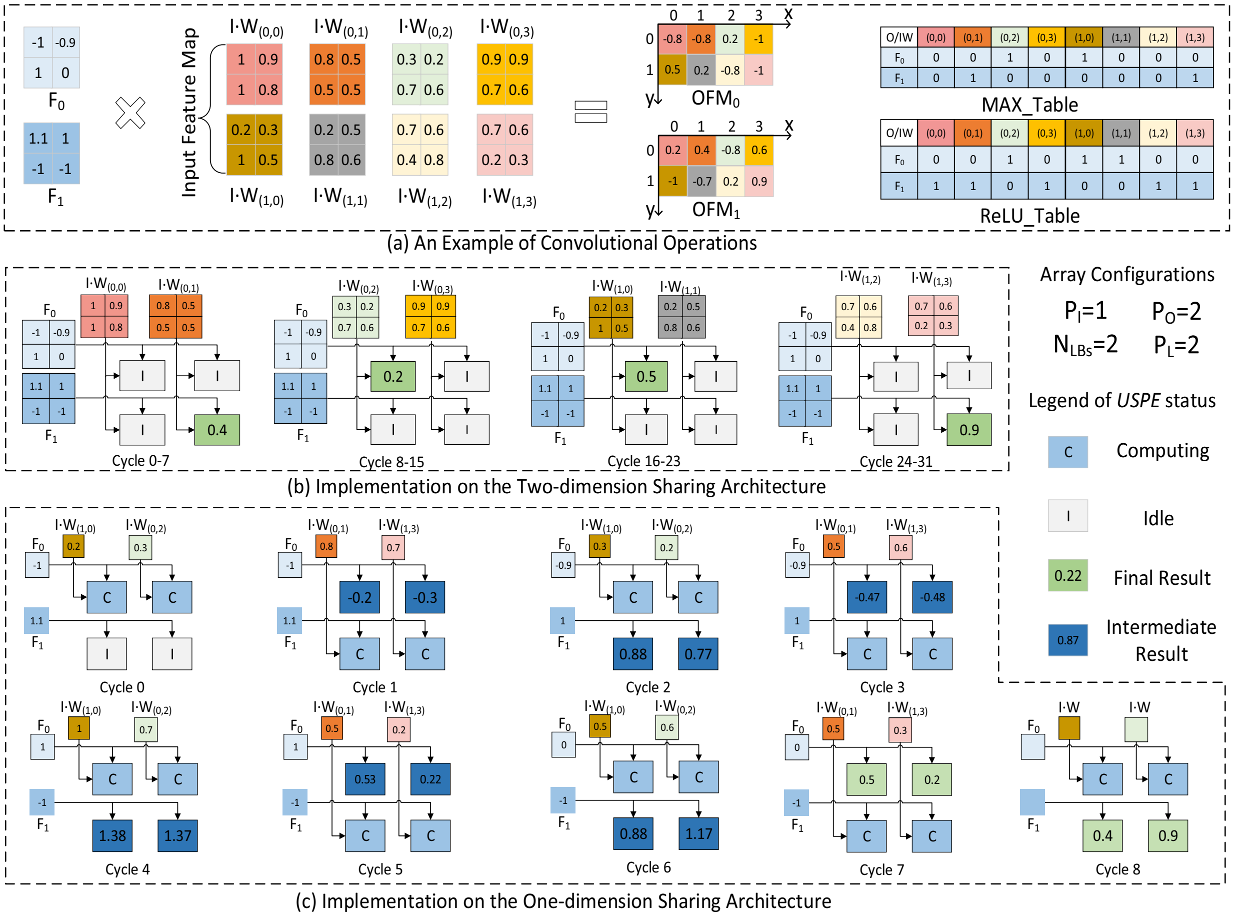
Figure 10. Scale-out Design in Prediction Stage

This uniform, serial multiplier based architecture can be used in both predictor and executor regardless of the bit-width of their inputs. The difference in the bit-width will be reflected by the number of running cycles. Take a 3-13 input splitting in 16-bit fix-point format as an example. It takes 3 cycles for prediction and 13 cycles for execution. Thus, with the serial multiplier, the reduced bit-width can lead to lower latency.

In Fig. 7, we deploy the serial multiplier into our two-stage based processing element architecture (*USPE*). Compared with the design shown in Fig. 5, the predictor and the executor share a uniform serial multiplier based architecture. This uniform architecture overcomes the above two disadvantages: demanding multiple multiplier designs and pipeline bubbles. We use a Finite State Machine (*Stage* Signal) to control the computation conversion between Prediction Stage and Execution Stage. When the current state is Prediction Stage, I_{HBs} is enabled and the *USPE* is configured as the predictor. Using the *Sign* unit, the prediction results can be converted to 1 (positive) and 0 (negative) and then wrote into *ReLU_Table*. Using *Comparator* unit, the prediction results can be used to find the maximum output neurons, which are recorded in *MAX_Table*. We will describe details of these tables in Section IV-B. At Execution Stage, *USPE* only processes the EONs based on *ReLU_Table* or *MAX_Table*. I_{LBs} related to effectual output neuron is enabled and the remaining calculation is then performed on the uniform architecture. Finally, the final values of EONs are obtained by adding the results of Prediction Stage.

IV. OVERALL SCALE-OUT DESIGN

In Section III we propose *USPE* as a basic processing architecture that is able to bypass the computation of



O/IW	(0,0)	(0,1)	(0,2)	(0,3)	(1,0)	(1,1)	(1,2)	(1,3)
F_0	0	0	1	0	1	0	0	0
F_1	0	1	0	0	0	0	0	1

MAX_Table

O/IW	(0,0)	(0,1)	(0,2)	(0,3)	(1,0)	(1,1)	(1,2)	(1,3)
F_0	0	0	1	0	1	1	0	0
F_1	1	1	0	1	0	0	1	1

ReLU_Table

(a) An Example of Convolutional Operations

(b) Implementation on the Two-dimension Sharing Architecture

(c) Implementation on the One-dimension Sharing Architecture

Figure 11. Mapping MaxP to the Scale-out Design

ineffectual output neurons (iEONs) at single neuron level. However, since DNN computation usually involves a large number of neurons, the scalability of *USPE* is even more crucial.

We present a simple scale-out design as shown in Fig. 10, where the *USPE* unit is duplicated in two dimensions to improve the processing throughput. Since *USPE* uses a bit-serial multiplier to perform the basic multiplication, it takes multiple cycles to accomplish one multiplication. In the worst case, *USPE* can take maximum N_{LBs} *(max*N_{LBs}*)* cycles to calculate a product in Execution Stage. To maintain the comparable performance as obtained by a parallel multiplier, we need to simultaneously process P_L output neurons in one output feature map (OFM), where P_L is larger than $maxN_{LBs}$. Therefore, the parallelism granularity of the horizontal direction in Fig. 10 is P_L. To match the processing throughput of other state-of-the-art accelerators, such as DianNao, we also unroll various filters so that our accelerator can simultaneously perform multiple OFMs. As shown in Fig. 10, the vertical parallelism granularity is the number of OFMs (P_O). In this section, we explore the challenges of integrating our two-stage based computing pattern to the scale-out architecture and propose our solutions.

A. The Impact of Scale-out on Prediction Stage

We first examine whether the prediction stage could be seamlessly implemented on the scale-out design without introducing overheads. As shown in Fig. 10, each *USPE* is responsible for the calculation of one output neuron. Thus, this *USPE* array can predict P_L×P_O output neurons in parallel. There exist two opportunities for data sharing (the fetched data could be shared among *USPE*s after only one memory load) in this *USPE* array, which could be leveraged by Prediction Stage to mitigate memory access overhead. The first is filter sharing. Since all output neurons in one row belong to the same OFM and each OFM corresponds to one dedicated filter, the *USPE*s in one row can share the same filter data. The second is input sharing. Although the output neurons that are processed in the same column belong to different OFMs, they are located at the same place of their OFMs. Note that the output neurons with the same location are calculated from the same input neurons. Thus, all the *USPE* in one column share the same input data. (e.g., As shown in Fig. 10, though all green neurons belong to OFM₀, OFM₁, … OFM$_{P_O}$, respectively, they all locate at the top left corner of each output feature map and are calculated using the same input window (IW$_0$)). Since all the output neurons need to be predicted in Prediction Stage, these two sharing opportunities can be fully exploited. With the two kinds of data sharing (input and filter), P_L inputs and P_O filters can make the *USPE* array be fully-utilized.

After the prediction in Prediction Stage, we can obtain the location information of EONs. And the prediction results are written into the *Max_Table* and *ReLU-Table* respectively in Fig. 11(a). Each element in these tables corresponds to one output neuron. Value 0 means its corresponding output neuron is ineffectual, while 1 indicates effectual.

B. Challenges and Optimizations of Execution Stage

We then examine the challenges of parallelizing Execution Stage in the scale-out design. Through analyzing the randomness, we manage to find the underlying regularities of the number and the distribution of EONs in MaxP and ReLU respectively. We then propose corresponding data sharing policies for MaxP and ReLU and validate our design.

1) Challenges Caused by Randomness and Sparsity of EONs

Randomness of EONs: Since the iEONs could be bypassed based on prediction results in Prediction Stage, we only need to perform the remaining calculation related to the EONs recorded in the *Max_Table* and *ReLU_Table*. However, the distribution of EONs in OFMs is irregular. In MaxP layers, the distribution of EONs is random in each OFM. For instance, as shown in Fig. 11(a), the EONs (i.e. maximum output neurons) are located at (0,2) and (1,0) in OFM$_0$, while they are located at (0,1) and (1,3) in OFM$_1$. This random distribution also applies to ReLU. Besides distribution, the number of EONs in each OFM is different as well in ReLU. For example, as shown in Fig. 11(a), the number of EONs in OFM$_0$ is 3, while OFM$_2$ has 4 EONs.

Idleness of USPE Array: The naïve scale-out design shown in Fig. 10 does not well-support iEON skipping. Since the filter and input data are shared among the same row or column of *USPE*s, the *USPE* has to be set as idle when it is assigned an iEON while its neighbor is assigned an EON. With the naïve design, the number of idle *USPE*s equals to the number of iEONs. For example, 75% of *USPE*s are idle for MaxP as shown in Fig. 11(b). Consequently, the approach of prediction will merely contribute to the power saving of accelerator by power-gating idle *USPE*s, while the significant benefit to reduce the execution time will not be obtained in this naïve architecture. To improve the performance of Execution Stage, we should effectively bypass the process of iEONs according to *MAX_Table* and *ReLU_Table*, and enable the *USPE* array to perform EONs without idleness.

2) Data Sharing Policy in Execution stage

One method to increase the utilization of *USPE* is not to leverage the sharing opportunities as mentioned in Section IV-A. Without sharing the input data and filter within peer *USPE*s in the same row or column, each *USPE* can load its own input and filter on-demand based on its assigned EON. Although this method can avoid the idleness of *USPE*, it also challenges the memory access capacity. Without the input and filter sharing, we need $P_L \times P_O$ filters and inputs to keep all the *USPE*s busy. Since the EONs are randomly located in the OFMs, all their requisite inputs and filters are also randomly located in the input and filter memory so that they cannot be fetched together. In the worst case, it requires $P_L \times P_O$ cycles to load all the data, while *USPE* array only needs N_{LBs} cycles to process these data. Since the data loading time ($P_L \times P_O$) is larger than the data processing time (N_{LBs}), memory access becomes the bottleneck of the accelerator.

TABLE II. THE NUMBER OF EONS

Layer	Per Coordinate		Per OFM	
	Mean	Var	Mean	Var
Alex Layer1	40	4	1884	597
VGG11 Layer2	77	7	7579	2767

To alleviate the pressure of memory access, we still need to adopt one kind of data sharing in Execution Stage. There are two choices (input or filter sharing) for MaxP and ReLU.

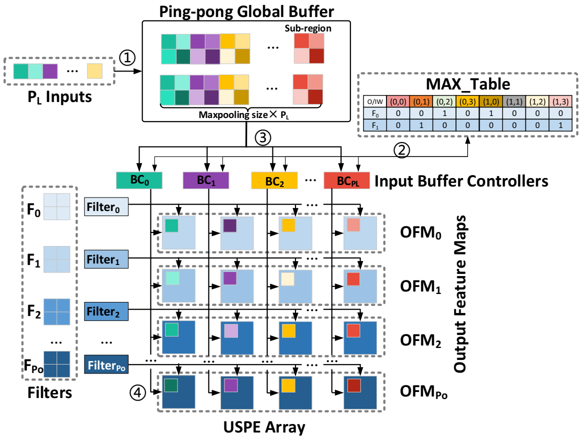
Max-pooling: Although the distribution of EONs is random in different OFMs, the number of the EONs in each OFMs is nearly the same and can be calculated based on the network architecture parameters, such as max-pooling size and stride size. For instance, as shown in Fig. 11(a), the percent of EONs in one OFM is about 25% for the 2×2 MaxP layer with 2 stride size. The stable number of EONs among OFMs indicates that the calculation time of each OFM is similar to others. Moreover, all the EONs in the same OFM correspond to the same filter. Therefore, we choose filter sharing for MaxP and the idleness can be avoided since the computation time of different OFMs is balanced.

***ReLU*:** Different from the MaxP, the number of EONs in each OFM varies for ReLU. If we still choose filter sharing for ReLU, the *USPE*s in one column may have different workloads, which results in the idleness of *USPE* when its workload is light. Fortunately, the summation of the EONs with the same coordinate across different OFMs is nearly the same. For example, as shown in TABLE II, the variance only accounts for 10% of its mean for the number of EONs with the same coordinate. Since output neurons belonging to the same location is calculated from the same input, we choose input sharing for ReLU.

3) Max-pooling: Filter Sharing

Since we use filter sharing for MaxP, the memory access related to input becomes the main bottleneck in Execution Stage. To tackle this challenge, we design P_L input buffer controllers as shown in Fig. 12. Each buffer controller is responsible for assigning the input data to the *USPE*s in one column.

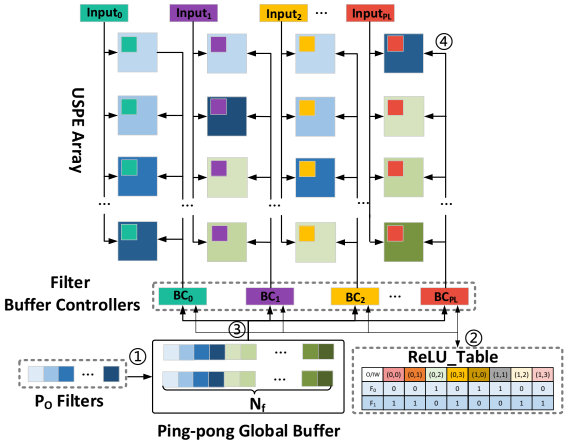
After the input data is loaded from memory, it will be cached in a Ping-pong Global Buffer (Step ①: data loading). This global buffer is built upon a basic idea of double-buffering, in which double buffers are operated in a ping-pong manner to overlap data loading with computation. Then, according to the *MAX_Table* records (step ②), buffer controllers distribute the input data to their corresponding *USPE*s in the following cycles (step ③: data assignment).



MAX_Table

O/IW	(0,0)	(0,1)	(0,2)	(0,3)	(1,0)	(1,1)	(1,2)	(1,3)
F_0	0	0	1	0	1	0	0	0
F_1	0	1	0	0	0	0	0	1

Figure 12. Scale-out Design for MaxP



ReLU_Table

O/IW	(0,0)	(0,1)	(0,2)	(0,3)	(1,0)	(1,1)	(1,2)	(1,3)
F_0	0	0	1	0	1	1	0	0
F_1	1	1	0	1	0	0	1	1

Figure 13. Scale-out Design for ReLU

To keep all the *USPE*s busy, the above design should meet the following requirements so that it can overcome the bottleneck of memory access. First, the time of data assignment should be shorter than that of data processing. Second, the time of loading data to global buffer also should be shorter than that of data processing. Each *USPE* takes the bit-width of input (N_{LBs}) cycles to finish the data processing using the serial multiplier. We explore the detailed time model of data assignment and loading and validate that our design satisfies the above requirements.

Data Assignment: For MaxP layers, there must exist at least one EON in each sub-region with the size of *maxpooling_size* (e.g. 2×2 in Fig. 12) in the OFM. Therefore, we assign one sub-region to a *USPE*. The *USPE* calculates the EON in this region. In the worst case, all locations within the sub-region can have the EON. In this situation, all *USPE*s in the same column need *maxpooling_size* inputs. Since buffer controller can only broadcast one input each time, it will take *maxpooling_size* cycles to finish the input assignment. Normally, the *maxpooling_size* is small, such as 2×2 or 3×3. Based on TABLE I, the N_{LBs} is always bigger than 12. Therefore, the time of input assignment (*maxpooling_size*) is shorter than that of data processing (N_{LBs}), which indicates the memory controller design is capable of keeping all the *USPE*s busy.

Data Loading: Since all *USPE*s in one column needs *maxpooling_size* inputs under the worst case, the *USPE* array with P_L columns demands at most *maxpooling_size* × P_L inputs, whose size is also the size of cached input in the global buffer. As the global buffer can load P_L consecutive inputs each time, it needs *maxpooling_size* cycles to accomplish the data loading. Since N_{LBs} is bigger than *maxpooling_size*, with the ping-pong buffer design, the data loading time can be overlapped with the time of data processing.

We take a simple example (two filters convolve with eight input windows, the output consists of two OFMs and each has 2×4 output neurons, the number of processing cycles (N_{LBs}) is 2) in Fig. 11(a) to illustrate our design. After MaxP layer, each OFM only has two EONs as shown in *MAX_Table*. In Fig. 11(b), due to the two-dimension sharing, this *USPE* array could not skip over the iEONs, which results in 75% idle *USPE*. Fig. 11(c) only keeps the filter sharing among *USPE* in one row and introduces buffer controllers to assign the input data. With the buffer controller, all the *USPE*s can obtain its input within two cycles. Considering the two-cycle processing time (N_{LBs}), all the *USPE*s can obtain its new input data when finishing the processing of current input data. Thus, this *USPE* array can be fully utilized after cycle 0. Since the calculation of output neuron includes four MAC operations and one multiplication needs two cycles using the serial multiplier, this *USPE* array takes 8 (2x4) cycles to finish all the processing.

4) *ReLU: Input Sharing*

Since we use input sharing for ReLU, the filter-related memory access becomes the main bottleneck. Here we reuse the ping-pong global buffer and the buffer controllers as described in Section IV-B3 to overcome this bottleneck. Compared with the design for MaxP, all the *USPE*s in one column share the same input and each buffer controller is responsible for filter distribution among *USPE*s belonging to one column shown in Fig. 13.

Similarly, to fully-utilize the *USPE* array, our design must meet the requirements of data assignment and loading discussed in Section IV-B3. Next, we discuss the time of data assignment and loading for ReLU.

Data Assignment: Since each buffer controller is responsible for P_O *USPEs* and each *USPE* requires a different filter under the worst case, it takes P_O cycles to finish the data assignment. To ensure memory controller can provide enough filters for each *USPE*, the cycles of data assignment should be less than that of data processing as (3):

$$P_O \leq N_{LBs} \quad (3).$$

Data Loading: After the prediction in Prediction Stage, we obtain the ratio of EONs (α) belonging to the same output location. To keep all the *USPE*s busy, the effectual number of filter ($\alpha \times N_f$) should be larger than the parallelism of filters (P_O), where N_f is the number of filters we need to store in the global buffer. Therefore, N_f can be calculated as (4):

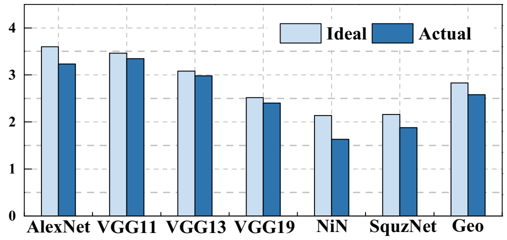


Figure 14. Speedup of CONV among Different Networks

$$N_f \geq \frac{P_O}{\alpha} \quad (4).$$

Since the global buffer can load P_O consecutive filters at one time, it needs $1/\alpha$ cycles to finish the filter loading. As α is normally larger than 10% and N_{LBs} is larger than 12, the time of data loading $(1/\alpha)$ is less than that of data processing, which demonstrates it can meet the requirement of data loading.

C. Fully Connected Layer

In FCN layers, each output neuron has each own filter. Since each *USPE* is responsible for one output neuron, each *USPE* needs to load its own filter and there is no filter sharing in the *USPE* array. Therefore, filter access is a bottleneck for FCN layers. To keep all the *USPE*s busy, we need to load $P_L \times P_O$ filters. In Execution Stage, under the worst case, the EONs are randomly located in the OFM, which means their corresponding filters are also randomly located in memory. It is impossible to randomly load $P_L \times P_O$ filters within N_{LBs} cycles. To alleviate the pressure of memory access, we add an accumulator to each row and make the *USPE*s in one row responsible for one output neuron. Each time we load a filter related to EONs and assign its weights to the *USPE*s in one row. Since the weights belonging to a filter are consecutively located in memory, we can load them within one cycle. In this way, it takes P_O cycles to finish the filter access. After choosing P_O that satisfies (3), we can tackle the challenge of filter access and make the *USPE* array being fully-utilized.

V. Evaluation

In this section, we first introduce our methodology. Then, we evaluate the performance improvement and energy efficiency of our design. We also break down the performance improvement and explore the trade-off between global buffer size and speedup. Finally, we compare our design with other state-of-the-art schemes, such as *Cnvlutin* [21] and *Stripes* [23]. We further combine our design with *Cnvlutin* and *Stirpes* respectively to demonstrate the improvement introduced by our method.

A. Methodology

The evaluation uses the set of popular, and state-of-the-art convolutional neural networks [1], [4], [25], [31], [32] shown in TABLE I. The networks are pre-trained using Pytorch [33]. To keep the accuracy same as the baseline with 16-bit, we use the number of high-order bits (N_{HBs}) in TABLE I to perform Prediction Stage and the remaining bits (16 - N_{HBs}) to perform Execution Stage.

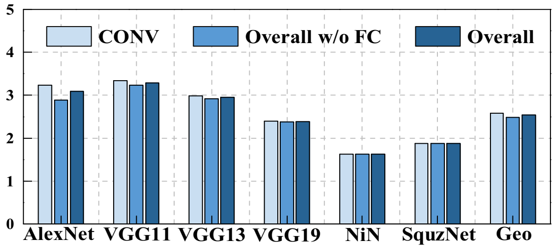


Figure 15. Overall Speedup among Different Networks

Our baseline is the MIFM-MOFM architecture. We choose the same parallelization configuration (P_I=16 and P_O=12) for our design and the baseline. Our *USPE* is based on a serial multiplier, while baseline's processing element is implemented with a 16-bit parallel multiplier. To maintain the comparable performance as obtained by a parallel multiplier, we simultaneously process 16 output neurons in one OFM (P_L=16). We implement our design and baseline using Xilinx VCU118 Evaluation board. It includes two sets of five 512MB DDR4 SDRAM and a Xilinx UltraScale+ XCVU9P FPGA.

B. Performance Improvement

Fig. 14 illustrates the speedup of CONV layers across different DNNs relative to the baseline. In Fig. 14, the ideal speedup is calculated under the ideal case where no idleness exists, and the real speedup is achieved by our hardware implementation. In general, our prediction-based design yields an ideal speedup of 2.8X, and an actual speedup of 2.6X, on average, across all DNNs over the baseline. This illustrates that the actual speedup is approaching the ideal speedup by our idleness-reduction design in Section IV-B. It also shows that we exploit the inherent predictability of DNNs to remove nearly all the multiplications related to the iEONs.

Fig. 15 evaluates the overall performance of our design on both convolution layers and FCN layers. Without FCN optimization, the overall speedup will degrade compared to CONV speedup. With our optimization for FCN layers, the overall speedup will be improved slightly for AlexNet and VGG11. For other networks where FCN layers accounts for very small proportion of total computations, their performance is mainly determined by CONV layers and the overall speedup nearly keeps fixed. Although our FCN optimization has a limited improvement on performance, it will greatly mitigate the memory access overhead and reduce reasonable energy, which will be illustrated in Section V-D.

C. A Breakdown of Performance Improvement

In this section, we illustrate the breakdown of performance improvement. Compared with the naïve design in Section II-C, our optimization includes computation reuse and architecture reuse. As shown in Fig. 16, although the naïve design can skip over the iEONs, it only has 1.2X speedup on average, which is far from ideal speedup. This is because the extra overheads (including area overhead and

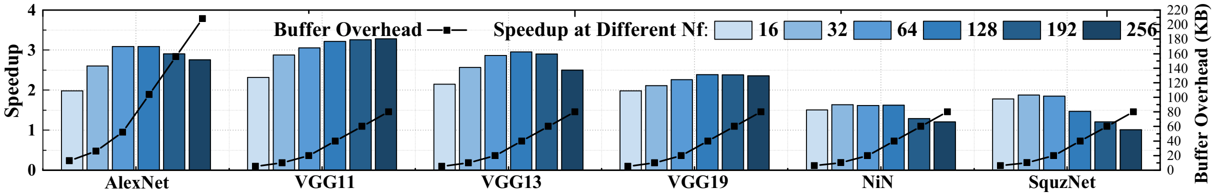


Figure 18. Trade-off between Buffer Size and Speedup

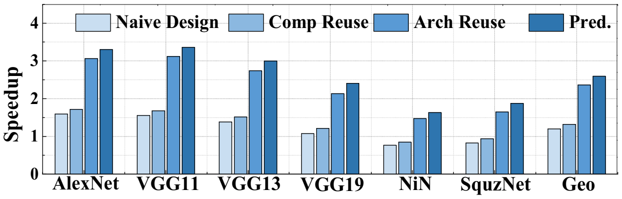


Figure 16. Breakdown of Performance Improvement

computation overhead) introduced by prediction. For some networks, such as NiN, the overheads even offset its benefit. Compared with architecture reuse, computation reuse contributes little improvement on performance. This is because the benefit of computation reuse is limited by the number of high-order bits in Prediction Stage. To reduce the overhead of predictor, we try to minimize the number of bits in Prediction Stage, which will reduce the benefit of computation reuse.

Instead of dedicated architectures for predictor and executor, our uniform architecture is shared by predictor and executor at different time slots. This architecture reuse can significantly improve the performance by avoiding the idleness of dedicated architectures where one architecture is waiting for the other.

D. Energy Efficiency

In this section, we compare energy efficiency w/ and w/o FCN optimization. As shown in Fig. 17, without considering off-chip data access, our design has an average 2.7X energy efficiency and the energy efficiency is nearly the same even with FCN optimization. This is because the computations in CONV layers dominate the energy consumption without off-chip data access. And our prediction based design can greatly remove the computations in CONV layers by skipping over all the computations related to iEONs.

When considering the off-chip data access, the FCN optimization (weight splitting) plays an important role in energy efficiency, especially for the networks with large amount of filter weights in FCN layers, such as AlexNet and VGGNet. Our FCN optimization can reduce the off-chip memory access by avoiding loading the remaining bits of filter weights related to iEONs in Execution Stage and the decrease in off-chip memory access further contributes the high energy-efficiency. As shown in Fig. 17, FCN optimization improves the average energy-efficiency from 1.5X to 1.9X.

E. Trade-offs among Speedup, Idleness, and Overhead

Due to the limited memory bandwidth, we could not load all the filters (total number is *nF*) into the global buffer within N_{LBs} cycles. Therefore, each time we only load one group of filters and the group size is N_f. As discussed in Section IV-B4, not only is the value of N_f related to global buffer size, but also it influences the idleness of *USPE* array. Thus, in Fig. 18, we demonstrate the relationship between global buffer size and speedup with various N_f. Note that the global buffer size is proportional to N_f since all the loaded filters need to be stored in the global buffer. For the speedup, it first increases then plateaus as N_f continues to increase. At the beginning, since the N_f is small, it could not satisfy (4), which results in the idleness of *USPE* array. With increasing N_f, the idleness will be mitigated. And N_f with 64 nearly makes this *USPE* array being fully-utilized. Thus, after N_f is bigger than 64, the speedup does not continue to increase. For SqueezeNet, the speedup even decreases when N_f is larger than 64. This is mainly caused by its small number of filters (*nF*) in most of the layers. If the group size (N_f) is bigger than *nF*, some of *USPE*s become idleness due to lack of filters, which results in performance degradation of SqueezeNet. To sum up, the performance improvement comes at the expense of global buffer size and the performance not always increase. We should consider this trade-off during accelerator design.

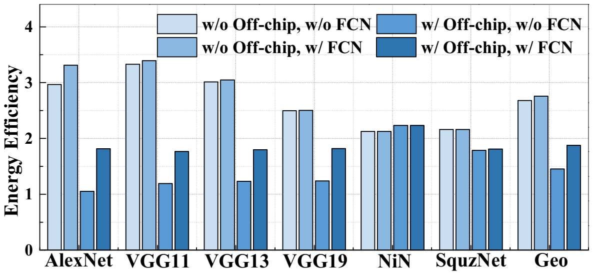


Figure 17. Energy-Efficiency Comparison w/ & w/o FCN Optimization

F. Benefits for Other Works

By leveraging the idea of prediction in DNNs, our accelerator design achieves performance and energy efficiency improvement. More importantly, our prediction idea is orthogonal to other state-of-the-art works, such as *Cnvlutin* and *Stripes*, and can provide the community with more opportunities to exploit. Fig. 19 first compare our design with *Cnvlutin* and *Stripes*, then demonstrates what will happen if we integrate these ideas together. The baseline is still the MIFO-MOFM architecture.

In Fig. 19, our design can achieve 1.8X speedup on average over *Cnvlutin*. This is because our prediction-based design can remove more ineffectual multiplications as shown in Fig. 1. Different from exploiting the sparsity of DNNs as *Cnvlutin* and our design, *Stripes* leverages the variance in the numerical precision requirements of DNNs to improve the performance. Thus, *Stripes* needs to perform the calculations

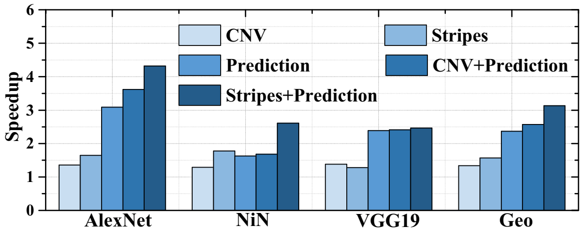


Figure 19. Comparisons with *Cnvlutin* (*CNV*) and *Stripes*

related to all the output neurons, while our method only needs to process the EONs. Compared with *Stripes*, our method has an average 1.6X speedup.

Next, we combine our prediction-based design with *Cnvlutin* and *Stripes* respectively to measure the improvement introduced by our method. This design combination (*Prediction*+*Cnvlutin*) achieve an average 1.9X speedup over *Cnvlutin* because our design can further remove non-zero-valued operand multiplication related to iEONs. Since our characterization in TABLE I demonstrates the prediction (Prediction Stage) needs the less precision (N_{HBs}) than that of *Stripes* ($N_{stripes}$), our prediction-based method can be utilized in *Stripes*. With the combination method (*Prediction*+*Stripes*), in Prediction Stage, we use N_{HBs}-*width* input for prediction. In Execution Stage, we can perform the calculation using the remaining bits ($N_{Stripes}$ - N_{HBs}). Since this combination design can skip over the calculations related to iEONs, it has an average 2.0X speedup over *Stripes*. Therefore, the output sparsity discovered by our prediction-based method is a good opportunity to further improve the performance of other state-of-the-art works.

VI. Related Work

The computational requirements and applicability of deep neural networks have prompted researchers to design numerous proposals for hardware acceleration, with implementations on either FPGAs [13], [28], [34] [27], [29], [35] or ASICs [10]–[12], [14]–[16], [21], [30], [36]–[38]. *DianNao* [30] and *DaDianNao* [10] utilize a large global buffer as a shared storage to reduce DRAM access energy consumption. Farabet et al. [36] propose a systolic architecture called *NeuFlow* where filter weights remain stationary in the register to maximize their reusability. To achieve high-energy efficiency, *Eyeriss* [16] proposes a row stationary dataflow by exploiting local data reuse of filter weights and input neurons. All the above accelerators mainly focus on reducing energy consumption via memory access optimization. Besides achieving high-energy efficiency, our accelerator can significantly improve performance with the novel prediction-based two-stage computing architecture.

The sparsity of DNNs opens a new opportunity to optimize the performance of accelerators. Nevertheless, most of the existing efforts (e.g. *Cnvlutin* and *Eyeriss*) mainly focus on the sparsity in filter weights and input feature maps [11], [14], [17], [21]. For example, *Eyeriss* can gate zero input neuron computations to further save power. Instead of powering off the zero neuron computations, *Cnvlutin* directly skips over the zero inputs. Different from *Cnvlutin*, we skip over all the computations related to the iEONs. *EIE* [11] can accelerate the processing of the networks by skipping the zero inputs and weights. However, it limits to the fully connected layers of a CNN model, while our work can be applied to both convolutional layers (the majority of the computations) and fully connected layers. *Scalpel* [39] proposes node-pruning technique to remove redundant nodes in DNNs by using mask layers to dynamically identify and remove unimportant nodes. Note that the node in *Scalpel* indicates an output feature map. If the node is unimportant, all the output neurons in this node (output feature map) will be removed. Our work only removes the iEONs in each output feature map. Since the pruning in *Scalpel* is more aggressive, it needs to retrain the network to restore the accuracy, while our work can achieve the same accuracy as the baseline without retraining.

VII. Conclusion

This work first introduces the prediction-based idea to DNN accelerator design. By predicting the iEONs of CONV and FCN in advance, our accelerator can significantly eliminate the ineffectual computation and reduce memory access. To minimize the overhead of predictor, we propose a uniform architecture *USPE* for predictor and executor. Our *USPE* can also leverage the computation reuse between predictor and executor to reduce the computational overhead. To improve the processing throughput, we present scale-out design for *USPE*. And our novel scale-out design can reduce the idleness of *USPE* due to the randomness and sparsity of EONs. Evaluation results show that our proposed design achieves an average 2.5X speedup and 1.9X energy-efficiency over the traditional accelerator. Our prediction-based design has an average 1.8X speedup over *Cnvlutin* and 1.6X speedup over *Stripes*. Moreover, when combined with our design, *Cnvlutin* and *Stripes* can be improved by 1.9X and 2.0X on average respectively.

Acknowledgment

This work is supported in part by NSF grants 1527535, 1423090, 1320100, 1117261, 0937869, 0916384, 0845721 (CAREER), 0834288, 0811611, 0720476, by SRC grants 2008-HJ-1798, 2007-RJ-1651G, by Microsoft Research Trustworthy Computing, Safe and Scalable Multicore Computing Awards, and by three IBM Faculty Awards.

References

[1] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In Advances in Neural Information Processing Systems (NIPS), 2012.

[2] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition, arXiv Prepr. arXiv1512.03385v1, 2015.

[3] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks, arXiv Prepr. arXiv1506.01497, Jun. 2015.

[4] A Simonyan, K. and Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition, CoRR, vol. abs/1409.1, 2014.

[5] Mingcong Song, Yang Hu, Huixiang Chen, and Tao Li. Towards Pervasive and User Satisfactory CNN across GPU Microarchitectures. In 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA), 2017.

[6] Mingcong Song, Yang Hu, Yunlong Xu, Chao Li, Huixiang Chen, Jingling Yuan, and Tao Li. Bridging the Semantic Gaps of GPU Acceleration for Scale-out CNN-based Big Data Processing: Think Big, See Small. In The 25th International Conference on Parallel Architectures and Compilation Techniques （PACT）, 2016.

[7] How AI is Making Self-Driving Cars Smarter: http://www.roboticstrends.com/article/how_ai_is_making_self_driving_cars_smarter.

[8] Deep Learning For Smart Cities: https://prateekvjoshi.com/2016/07/05/deep-learning-for-smart-cities/.

[9] Mingcong Song, Kan Zhong, Jiaqi Zhang, Yang Hu, Duo Liu, Weigong Zhang, Jing Wang, and Tao Li. In-situ AI: Towards Autonomous and Incremental Deep Learning for IoT Systems. In 2018 IEEE International Symposium on High Performance Computer Architecture (HPCA), 2018.

[10] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. DaDianNao: A Machine-Learning Supercomputer. In ACM/IEEE 47th Annual International Symposium on Microarchitecture (MICRO), 2014.

[11] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark Horowitz, and Bill Dally. EIE: Efficient Inference Engine on Compressed Deep Neural Network. In ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA), 2016.

[12] Brandon Reagen, Paul Whatmough, Robert Adolf, Saketh Rama, Hyunkwang Lee, Saekyu Lee, Jose Miguel Hernandez Lobato, Gu-Yeon Wei, and David Brooks. Minerva: Enabling Low-Power, High-Accuracy Deep Neural Network Accelerators. In ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA), 2016.

[13] Yufei Ma, Yu Cao, Sarma Vrudhula, and Jae-sun Seo. Optimizing Loop Operation and Dataflow in FPGA Acceleration of Deep Convolutional Neural Networks. In ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA), 2017.

[14] Shijin Zhang, Zidong Du, Lei Zhang, Huiying Lan, Shaoli Liu, Ling Li, Qi Gu, and Yunji Chen. Cambricon-X: An Accelerator for Sparse Neural Networks. In ACM/IEEE 49th Annual International Symposium on Microarchitecture (MICRO), 2016.

[15] Ali Shafiee, Anirban Nag, Naveen Muralimanohar, Rajeev Balasubramonian, John Paul Strachan, Miao Hu, R Stanley Williams, and Vivek Srikumar. ISAAC: A Convolutional Neural Network Accelerator with In-Situ Analog Arithmetic in Crossbars. In ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA), 2016.

[16] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks. In ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA), 2016.

[17] Han Song, Huizi Mao, and William J Dally. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. In International Conference on Learning Representations (ICLR), 2016.

[18] Xiangyu Zhang, Jianhua Zou, Xiang Ming, Kaiming He, and Jian Sun. Efficient and Accurate Approximations of Nonlinear Convolutional Networks, arXiv Prepr. arXiv1411.4229, Nov. 2014.

[19] Michael Figurnov, Dmitry Vetrov, and Pushmeet Kohli. PerforatedCNNs: Acceleration through Elimination of Redundant Convolutions, arXiv Prepr. arXiv1504.08362, Apr. 2015.

[20] Cheng Tai, Tong Xiao, Yi Zhang, Xiaogang Wang, and Weinan E. Convolutional neural networks with low-rank regularization, arXiv Prepr. arXiv1511.06067, Nov. 2015.

[21] Jorge Albericio, Patrick Judd, Tayler Hetherington, Tor Aamodt, Natalie Enright Jerger, and Andreas Moshovos. Cnvlutin: Ineffectual-Neuron-Free Deep Convolutional Neural Network Computing. In ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA), 2016.

[22] CS231n: Convolutional Neural Networks for Visual Recognition: http://cs231n.github.io/convolutional-networks/.

[23] Patrick Judd, Jorge Albericio, Tayler Hetherington, Tor M Aamodt, and Andreas Moshovos. Stripes: Bit-Serial Deep Neural Network Computing. In ACM/IEEE 49th Annual International Symposium on Microarchitecture (MICRO), 2016.

[24] Rectifier: https://en.wikipedia.org/wiki/Rectifier_(neural_networks).

[25] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2015.

[26] Convolutional neural network: https://en.wikipedia.org/wiki/Convolutional_neural_network.

[27] Yongming Shen, Michael Ferdman, and Peter Milder. Maximizing CNN Accelerator Efficiency Through Resource Partitioning. In Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA), 2017.

[28] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks. In ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA), 2015.

[29] Manoj Alwani, Han Chen, Michael Ferdman, and Peter Milder. Fused-Layer CNN Accelerators. In ACM/IEEE 49th Annual International Symposium on Microarchitecture (MICRO), 2016.

[30] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. DianNao: A Small-Footprint High-Throughput Accelerator for Ubiquitous Machine-Learning. In 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2014.

[31] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and, arXiv Prepr. arXiv1602.07360, Feb. 2016.

[32] Min Lin, Qiang Chen, and Shuicheng Yan. Network In Network, arXiv Prepr. arXiv1312.4400, Dec. 2013.

[33] Pytorch: http://pytorch.org/.

[34] Kalin Ovtcharov, Olatunji Ruwase, Joo-young Kim, Jeremy Fowers, Karin Strauss, and Eric S Chung. Accelerating Deep Convolutional Neural Networks Using Specialized Hardware, Microsoft Res. Whitepaper, pp. 3–6, 2015.

[35] M Song, J Zhang, H Chen, and T Li. Towards Efficient Microarchitectural Design for Accelerating Unsupervised GAN-Based Deep Learning. In 2018 IEEE International Symposium on High Performance Computer Architecture (HPCA), 2018.

[36] Clément Farabet, Berin Martini, Benoit Corda, Polina Akselrod, Eugenio Culurciello, and Yann LeCun. NeuFlow : A Runtime Reconfigurable Dataflow Processor for Vision. In IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops (CVPRW), 2011.

[37] Shaoli Liu, Zidong Du, Jinhua Tao, Dong Han, Tao Luo, Yuan Xie, Yunji Chen, and Tianshi Chen. Cambricon: An Instruction Set Architecture for Neural Networks. In ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA), 2016.

[38] Wenyan Lu, Guihai Yan, Jiajun Li, Shijun Gong, Yinhe Han, and Xiaowei Li. FlexFlow : A Flexible Dataflow Accelerator Architecture for Convolutional Neural Networks. In 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA), 2017.

[39] Jiecao Yu, Andrew Lukefahr, David Palframan, Ganesh Dasika, Reetuparna Das, and Scott Mahlke. Scalpel: Customizing DNN Pruning to the Underlying Hardware Parallelism. In Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA), 2017.