Hardware-Software Codesign of Weight Reshaping and Systolic Array Multiplexing for Efficient CNNs

[1]Jingyao Zhang, [1]Huaxi Gu, [2]Grace Li Zhang, [2]Bing Li and [2]Ulf Schlichtmann
[1]Xidian University, [2]Technical University of Munich
jingyao.zhang.xidian@foxmail.com, hxgu@xidian.edu.cn, {grace-li.zhang, b.li, ulf.schlichtmann}@tum.de

***Abstract*—The last decade has witnessed the breakthrough of deep neural networks (DNNs) in various fields, e.g., image/speech recognition. With the increasing depth of DNNs, the number of multiply-accumulate operations (MAC) with weights explodes significantly, preventing their applications in resource-constrained platforms. The existing weight pruning method is considered to be an effective method to compress neural networks for acceleration. However, weights after pruning usually exhibit irregular patterns. Implementing MAC operations with such irregular weight patterns on hardware platforms with regular designs, e.g., GPUs and systolic arrays, might result in an underutilization of hardware resources. To utilize the hardware resource efficiently, in this paper, we propose a hardware-software codesign framework for acceleration on systolic arrays. First, weights after unstructured pruning are reorganized into a dense cluster. Second, various blocks are selected to cover the cluster seamlessly. To support the concurrent computations of such blocks on systolic arrays, a multiplexing technique and the corresponding systolic architecture is developed for various CNNs. The experimental results demonstrate that the performance of CNN inferences can be improved significantly without accuracy loss.**

***Index Terms*—neural networks, systolic arrays, hardware-software codesign, efficient CNNs**

I. Introduction

In recent years, deep neural networks (DNNs) have been widely applied in various fields, e.g., image/speech recognition. In DNNs, there are a large number of multiply-accumulate (MAC) operations. When DNNs grow deeper for higher prediction accuracy, the number of MAC operations explodes to an extremely huge quantity that resource-constrained platforms cannot afford executing within an acceptable time.

To accelerate MAC operations in DNNs, systolic arrays are introduced as an attractive platform due to their high degree of concurrent computation and high data reuse rate. Recently, various state of the art hardware accelerators using systolic arrays or properties of systolic arrays have been proposed. TPU is the most well-known accelerator based on systolic arrays [1]. In TPU, weights are first fetched to the processing elements (PEs), and then the streamed input data are pushed into the systolic array to keep the systolic array busy, similar to pipelining in digital design. By modifying the interconnection between multipliers and adders inside PEs, a fast and scalable systolic array is designed in [2], [3]. To accelerate the inference of 3D convolutional neural networks (CNNs), a 3D systolic cube is proposed in [4], where PEs are connected via a 3D-cube Network-on-Chip, and the dataflow has been redesigned to support both 2D and 3D convolutions.

Besides hardware accelerators for DNNs, on the algorithm level, weight pruning has also been developed to compress neural networks for acceleration in recent years. Most weights of neural networks, especially DNNs, are so close to zero that there is no reduction in accuracy if they are pruned (set to zero). For example, 92.5% of weights can be pruned without accuracy loss for VGG-16 [5]. However, the resulting weight matrix is unstructured and exhibits irregular patterns. Implementing MAC operations with such irregular weight patterns on hardware platforms with regular designs, e.g., GPUs and systolic arrays, might result in an underutilization of hardware resources. To overcome the challenge described above, structured pruning methods [6]–[9] have been proposed to prune weights in shape-wise, vector-wise, kernel-wise, filter-wise and channel-wise ways. Although these structured pruning approaches outperform unstructured pruning when systolic arrays are deployed, they might not match the underlying data-reuse patterns of systolic arrays. Consequently, such methods might still result in an underutilization of computing resources of the systolic array.

To solve the underutilization problem described above, LODESTAR in [10] compresses weights of CNN models in a particular block shape, which can be efficiently supported by systolic arrays. Though there is no underutilization of the systolic array, the compression ratio decreases significantly. The column combining method in [11] combines complementary columns into one column after unstructured pruning to further reduce the number of weights and alleviate the underutilization of systolic arrays. However, this method might not be effective for some CNNs. In this case, the sparsity of the weight matrix still exists. In addition, extra hardware cost is required to select corresponding inputs for weights.

To exploit the full benefits of systolic arrays, we propose a hardware-software codesign framework to efficiently accelerate various CNNs on systolic arrays. The main contributions of this paper are summarized as follows:

1) **Row swapping for compact storing of weight matrix** is proposed to move the sparse weight matrix after unstructured pruning to form a dense cluster, which can be compactly stored and efficiently streamed from memory.
2) **Block selection to cover the dense cluster of weights** is proposed to map various weight blocks seamlessly on systolic arrays, so that such arrays can be fully utilized to achieve low latency for CNNs.
3) **Systolic array multiplexing** is developed to support various weight blocks required by a CNN model. This technique requires the insertion of multiplexers into the systolic array to facilitate the concurrent computations of various weight blocks on systolic arrays.
4) **A flexible systolic array structure** is proposed to support the inferences of various CNN models with low

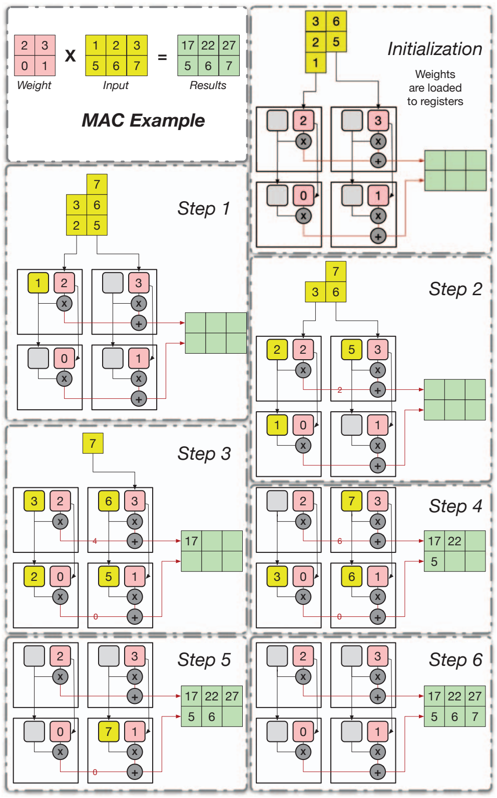

Fig. 1. Matrix multiplication with weight-stationary systolic array.

latency and low hardware cost. Since the locations of multiplexers for different CNN models vary significantly, to reduce the number of multiplexers, a genetic algorithm is deployed to select where the multiplexers should be inserted considering the hardware cost and latency on systolic arrays.

The rest of this paper is organized as follows. We first introduce the conventional systolic array for CNN inference and weight pruning for systolic arrays in Section II. The proposed hardware-software codesign framework is described in Section III. Experimental results are presented in Section IV. The conclusion is drawn in Section V.

II. Background and Motivation

A. Systolic Array

A systolic array includes a set of connected PEs, each having the same function. In such an array, data can flow directly between PEs in a pipelined fashion. To deploy such array for CNN inferences, a weight-stationary flow is usually adopted, where weights of neural networks are stored in the array without movement during computation, input data continuously enters into the array top-to-bottom and the computation result accumulates left-to-right. Since weights in convolutional neural networks are 3/4-dimensional, these high-dimensional weights should be unfolded into a 2-dimensional representation, so that they can be accelerated by systolic arrays.

Fig. 1 shows the process of matrix multiplication with a weight-stationary systolic array, where each PE consists of a multiplier, a full adder, and registers to buffer streamed data (full adder is not required for PEs in the first column). To achieve the multiplication of a weight matrix and an input matrix, weights are preloaded into PEs and then the inputs are streamed into PEs in a particular order. Each PE receives data from its upstream PE, stores data and passes data to its downstream PE. The intermediate product of a PE is transferred to the right PE for addition. After the addition of two products, the final result is stored in the appropriate position. To make the systolic array generate outputs in each cycle continuously, the input data should be properly skewed in the initialization as shown in Fig. 1, where the inputs 1 and 5 are not aligned.

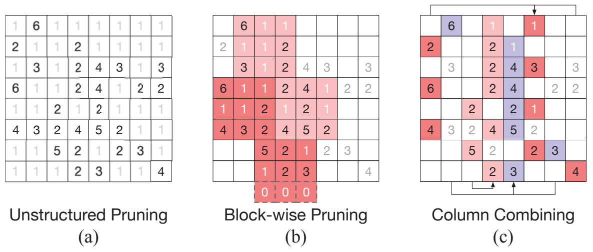

Fig. 2. Weight pruning for systolic arrays in previous work.

B. Weight Pruning for Systolic Arrays

Weight pruning aims to reduce the number of weights of CNNs by removing unimportant weights while still maintaining accuracy. An unstructured weight pruning method is to prune weights whose magnitudes are small, since the magnitude of an individual weight reflects the importance of the weight to some degree. It has been demonstrated that up to 90% weights in many neural networks can be pruned without any impact on classification accuracy. After the pruning, preserved weights spread around the weight matrix arbitrarily, and the distribution of non-zero weights is irregular. Structured pruning methods such as filter-wise pruning can prune weights in a specific pattern, they might have a low compression ratio (percent of pruned weights with respect to the number of original weights) compared with unstructured pruning.

The unstructured and structured pruning methods described above do not consider the implementation of MAC operations on systolic arrays, potentially leading to an underutilization of systolic arrays. To fully utilize systolic arrays, each PE should conduct valid computations for neural networks. To achieve this goal, previous work either prunes weights with a block whose size is the same as that of the systolic array or combines sparse columns of weights to form a block.

The concepts of the previous pruning methods for systolic arrays are illustrated in Fig. 2, where (a) shows a weight matrix after the unstructured pruning, with "1" in grey colour representing the weights that have been pruned with unstructured pruning. The results of the block-wise pruning method and the column-combining are shown in Fig. 2 (b)(c), with weights in grey colour representing that they are discarded for blocks and column combination. The block-wise method in Fig. 2(b) covers the sparse weight matrix with a block with a fixed size, so that many unimportant weights, shown as in white color, remain in these blocks. In addition, weights outside of the blocks are discarded directly. Therefore, it cannot provide a high compression ratio while maintaining the accuracy of

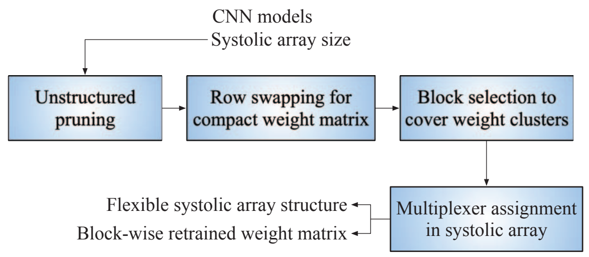
CNN models
Systolic array size
Unstructured pruning
Row swapping for compact weight matrix
Block selection to cover weight clusters
Multiplexer assignment in systolic array
Flexible systolic array structure
Block-wise retrained weight matrix

Fig. 3. Work flow of the proposed hardware-software codesign framework.

neural networks. The column-combining method in Fig. 2(c) squeezes the third column into the fourth column, the second and seventh column into the fifth column, and the first and the last column into the sixth column. However, many non-zero weights are discarded, potentially leading to a degradation of inference accuracy. In addition, this method incurs extra hardware cost for multiplexing inputs in the systolic array, since it breaks the vertical data dependence of the systolic array.

To maintain a high compression ratio and a high inference accuracy for neural networks while still avoiding the under-utilization problem of systolic arrays, a customized weight mapping method and dataflow for systolic arrays are required.

III. Hardware-Software Codesign Framework

To utilize the systolic arrays efficiently while guaranteeing a high compression ratio and high inference accuracy, we propose a hardware-software codesign framework considering both the latency and hardware cost of systolic arrays for CNNs.

The overview of the proposed framework is shown in Fig. 3. The inputs of the framework include given CNN models and the size of the systolic array, e.g., 64×64. At the software level, we first apply the unstructured pruning on the CNNs, since this method can prune a huge amount of weights. The resulting weight matrix is sparse. Afterwards, we swap rows in the weight matrix to achieve compact weight storing and efficient streaming from memory. The result of the row swapping is a dense weight cluster with an irregular shape. We then select various blocks to cover such an irregular weight matrix seamlessly. To achieve concurrent computations of various blocks on hardware implementations, multiplexers must be inserted on systolic arrays. To support these different pruned models with low latency and low hardware cost, the placement of multiplexers is generated by a genetic algorithm (GA). Finally, a flexible systolic array structure is produced to efficiently support the inferences of various CNNs.

A. Row swapping for compact storing of weight matrix

As mentioned in Section II.B, unstructured weight pruning achieves a high compression ratio but with the resulting weight matrix full of irregular patterns. An intuitive method to accelerate irregular weight patterns on systolic arrays with regular designs is to insert 0 into the weight matrix to recover the block-based weight matrix, so that the computation with such a weight matrix can be implemented on systolic arrays. However, this method incurs in huge memory consumption and an underutilization of systolic arrays since the multiplication with 0 in a lot of PEs is meaningless. The other method to accelerate irregular weight patterns is indexing each weight in

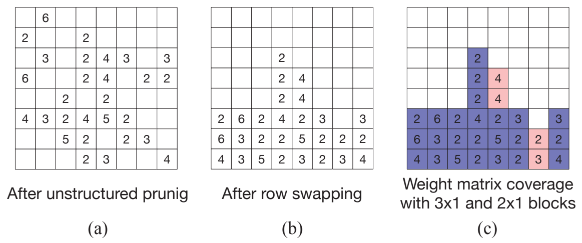
After unstructured prunig
After row swapping
Weight matrix coverage with 3x1 and 2x1 blocks
(a)
(b)
(c)

Fig. 4. (a) Weight after unstructured pruning; (b) Weight after row swapping; (c) Covering the weight matrix using different sizes of blocks, e.g., 3×1 and 2×1 blocks in this example.

the matrix with additional memory, leading to area overhead. In addition, a complicated controller is required to schedule the addition of the multiplication results of different weights, so that the latency of the systolic arrays to implement the multiplication of an input matrix and a sparse weight matrix is degraded significantly.

To overcome the challenges described above, we propose a row swapping method to store the sparse weight matrix densely. This method not only preserves the compression ratio of the weight matrix, but also requires smaller memory storage since only rows with non-zero weights should be indexed when each column has weights and a small number of columns should also be indexed when some columns are empty. The concept of this method is illustrated in Fig. 4 (a)(b), where the initial weight matrix after unstructured pruning is very sparse. After the proposed row swapping, all the remaining weights are pushed into the bottom of the weight matrix, forming a dense cluster. To store the weight matrix into the PEs of the systolic arrays, only the indexes of rows with non-zero weights are required. Therefore, the systolic array can accelerate the computations of the sparse weight matrix after row swapping.

Unlike the column combining in [11], the row swapping brings no extra hardware cost for multiplexing inputs in the systolic array, since it does not break the vertical data dependence of the systolic array, as shown in Fig. 1, where the inputs for each column of the weight matrix are the same on the systolic array. Another advantage of row swapping is that it makes the block selection feasible, which will be discussed in Section III.B.

B. Block selection to cover the dense cluster of weights

The weight matrix after row swapping forms a dense cluster with an irregular shape, as shown in Fig. 4(b). Therefore, it is difficult to find an $N \times N$, e.g., $N = 64$, block to cover this irregular weight cluster seamlessly for the efficient implementation on a systolic array with $N \times N$ PEs. To solve this problem, we propose to use different sizes of blocks to cover the reorganized weight matrix seamlessly. This concept is illustrated in Fig. 4(c), where the weights can be well covered with 3×1 and 2×1 blocks. The proposed method provides a finer granularity of coverage for the irregular weight matrix, so that no weights are discarded and there is no inference accuracy degradation.

When weight blocks with various sizes are mapped on systolic arrays to implement the matrix multiplication, the corresponding inputs for the weight blocks are not matched. For example, in Fig. 4(c), the last two column blocks need to

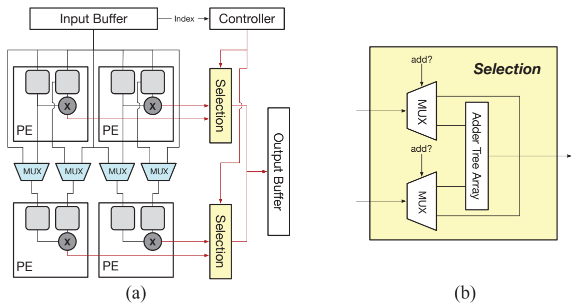

Fig. 5. (a) The modified systolic array structure with selection modules and multiplexers; (b) The structure of the selection module.

multiply with different input vectors. To support the concurrent computations of various blocks with the corresponding inputs on systolic arrays, we propose to modify the systolic arrays by inserting multiplexers for input selections of PEs. The modified systolic array is illustrated in Fig. 5(a). When the multiplexers are disabled, the systolic array behaves like the traditional one, where input data continuously enters into the array top-to-bottom and the computation result accumulates left-to-right. When the multiplexers are enabled, the systolic array is divided into small systolic arrays. The original inputs pass through only a part of PEs above the multiplexers vertically, while the new inputs are fetched from the buffer and enter the rest part of PEs at the same time. Since the original systolic array can be divided into several independent arrays by multiplexers, the vertical data dependence is partially eliminated, which increases the supported patterns of the systolic array.

Since the rows are swapped for the selection of various blocks to cover the weight matrix in the proposed method, the indexes of rows in the original weight matrix are required for accurate matrix multiplication. In addition, a controller is required to schedule the multiplication results. Therefore, we modify the systolic array architecture, as shown in Fig. 5(a). The controller receives indexes of weights, and determines whether the products from multiplying inputs with these weights should be added together, or directly stored in the output buffer. The control signals are transmitted to selection modules, as shown in Fig. 5(b). The selection module is responsible for the addition and storing the results into the right position of the output buffer. The original full adders (in Fig. 1) are merged from PEs into the adder tree array in the selection module.

The insertion of multiplexers results in area overhead. Consequently, the number of multiplexers should be as small as possible. Since this number is determined by the number of various blocks selected to cover the reorganized weight matrix, we restrict the number of different blocks to cover the weight matrix to be 2. In addition, the different locations of multiplexers lead to different latency of computation for systolic arrays. To determine the optimal block sizes to balance the latency and hardware cost, we minimize the following function:

$$E = C_l * L + C_a * A + C_w * W, \tag{1}$$

where E represents the evaluation result, L is the latency of the inference on systolic arrays, A is the area consumption of multipliers, full adders and multiplexers, W is the wiring cost resulting from systolic array multiplexing, and C_l, C_a, C_w are the coefficients of latency, area consumption, and wiring cost, respectively. Note that values of E, L and W here are normalized.

The detailed process of selecting the optimal block sizes for a given CNN is as follows:

1) Selecting p as the primary block size, where $p \in \mathbb{N}^+$ and $1 \leq p \leq 64$ (64×64 systolic array was used in the experiments);
2) Setting the secondary block size as q according to the mapping strategy where a simple linear combination with the given block size is used to fill the systolic array for efficient utilization. In this setting, $q = 64 \pmod p$;
3) Covering the weight matrix with the blocks with sizes $\{p \times 1, q \times 1\}$;
4) Evaluating the latency and hardware cost with the selected block sizes on the systolic array;
5) Iterating 1) - 4) until all 64 combinations of block sizes have been evaluated;
6) Choosing the combination of block sizes with the best tradeoff between latency and hardware cost.

After the evaluation with the method described above, the optimal block combination is generated for a given CNN to reduce its inference latency and hardware cost.

C. *Locating multiplexers in systolic array for various CNNs*

In different CNNs, the optimal combinations of block sizes vary significantly. Directly inserting multiplexers according to the various block sizes for different CNNs incurs high hardware cost. To balance the hardware cost and the inference latency for various CNNs, we propose to determine the locations of multiplexers with a genetic algorithm (GA) for efficient inference with systolic arrays.

The details of the genetic algorithm are described in Algorithm 1. The locations of multiplexers are encoded into a vector with binary values whose length equals $K \times R$, where K is the size of systolic array, e.g., 64, and R is the number of rows with multiplexers that are required to support different CNN models. Each individual in population is a binary vector of multiplexers. The tournament selection works by selecting each offspring as the one having minimal fitness in a chosen size, e.g., 2. The scheme of the crossover in Algorithm 1 is taken from the differential evolution algorithm. The crossover essentially selects a random point in the parent chromosome and inserts the partner values in each successive gene until the randomly generated number is higher than a certain probability. Then offsprings can be generated by the polynomial mutation:

$$x_i^{\{1,t+1\}} = x_i^{\{1,t\}} + \beta_i, \tag{2}$$

where $x_i^{\{1,t+1\}}$ is an offspring, $x_i^{\{1,t\}}$ is a parent, and β_i is calculated by the formula

$$\beta_i = \begin{cases} (2u_i)^{\frac{1}{\eta_u+1}} - 1, & u_i < 0.5 \\ 1 - [2(1-u_i)]^{\frac{1}{\eta_u+1}}, & u_i \geq 0.5 \end{cases} \tag{3}$$

where u_i is a random number in $[0, 1]$, and η_u is a user-defined non-negative real number.

TABLE I
EXPERIMENTAL RESULTS COMPARED WITH THE UNSTRUCTURED PRUNING METHOD

Dataset	Compression Ratio (%)		Accuracy (%)		Latency		#Multiplexers	
CIFAR-100	*Baseline*	*Proposed*	*Baseline*	*Proposed*	*Baseline*	*Proposed*	*Proposed*	*#Mult./#PEs (%)*
VGG-16	94.67	93.66	71.32	72.07	1	0.324	128	3.13
VGG-19	94.76	94.04	70.81	71.5	1	0.398	128	3.13
PreResNet-110	92.89	67.37	65.2	73.59	1	0.681	64	1.56
DenseNet-BC-40	90.99	75.11	67.27	71.95	1	0.291	192	4.69
DenseNet-BC-100	93.41	75.48	73.87	76.56	1	0.269	192	4.69

Algorithm 1: Genetic algorithm

```
START
1   Generate the initial population
2   Compute fitness: $E$
REPEAT
3   Tournament selection
4   Crossover
5   Polynomial Mutation
6   Compute fitness: $E = C_l * L + C_a * A + C_w * W$
UNTIL stopping criteria are satisfied
```

In the GA, the required combinations of blocks for different CNN models are set as the baseline. For example, if blocks, $\{26 \times 1, 12 \times 1\}$ and 32×1, need to be supported, *Row 26*, *Row 32* and *Row 52* of the baseline systolic array should be full of multiplexers according to the mapping strategy. After setting the baseline, some of the multiplexers are chosen by the GA to be removed. The evaluation process produces the corresponding latency and hardware cost for this removal of multiplexers for the current generation. Since we include the hardware cost in the evaluation function, the number of multiplexers decreases gradually until the evaluation value of latency and hardware cost converges. If the evaluation value is low enough or the algorithm reaches the maximal epochs, the GA will be terminated and provides the resulting locations of the multiplexers.

IV. EXPERIMENTAL RESULTS

The hardware-software codesign framework was implemented using PyTorch [12] on an Intel CPU with 3.4 GHz and an Nvidia Quadro RTX 6000 graphics card. The target CNN models are VGG-16, VGG-19, PreResNet-100, DenseNet-BC-40, and DenseNet-BC-100 on CIFAR-100 data set. The size of the systolic array is assumed to be 64×64. The unstructured pruning method used at the beginning of the framework is the method in [5]. The compression ratio is set as 95%. The baseline is the target CNN model that was pruned by the unstructured method [5]. The systolic array of the baseline is the same as the one of the proposed method, while the systolic array multiplexing is not enabled in the baseline. The latency in Table I is calculated by

$$L = \sum_{i=0}^{N} \max(\left\lceil \frac{T_p^i}{M_p^i} \right\rceil \times (p+D)+p, \left\lceil \frac{T_q^i}{M_q^i} \right\rceil \times (q+D)+q), \quad (4)$$

where p is the primary block size after block selection, q is the secondary block size, N is the number of convolutional layers in the given model, T_p^i, T_q^i is the total number of blocks with size p, q respectively in the i-th convolutional layer, M_p^i, M_q^i is the maximum number of blocks with size p, q supported by the modified systolic array respectively, D is the width of the 2-dimensional input matrix determined by general matrix-matrix multiplication (GEMM) [13]. We assume that loading data along with multiplication can be completed within one unit time.

The comparisons of compression ratio, inference accuracy, latency of CNNs on systolic arrays and the number of inserted multiplexers between the baseline and the proposed framework with different CNNs are demonstrated in Table I. The inference accuracies of all CNNs models with the proposed framework are higher than those after unstructured pruning, as shown in the third column in this table. The reason is that the proposed method recovers some of the originally pruned weights during the block selection to cover the weight matrix. The compression ratio, however, is reduced correspondingly compared with the baseline. For VGGs, the reduction is trivial, since the selected blocks are well suited to the reorganized weight matrix. The compression ratio for PreResNet-110, DenseNet-BC-40 and DenseNet-BC-100 is much lower than that of VGGs, since the coverage of weight matrix with various blocks is achieved by recovering many weights.

Though more weights are preserved to fulfill the blocks, thanks to the systolic array multiplexing, the total latency is reduced significantly, as shown in the seventh column. For example, latencies of DenseNet-BC-40 and DenseNet-BC-100 in the proposed design are 0.291, 0.269 of those in the baseline. This is mainly because the concurrency is improved significantly with systolic array multiplexing. The number of multiplexers to achieve systolic array multiplexing and the ratio between the number of multiplexers and the total number of PEs are shown in the last two columns. It is clear that the cost of inserted multiplexers is trivial for the total hardware cost.

Table II compares accuracy and compression ratio of the proposed method with the methods LODESTAR [10] and Column Combining [11]. Since LODESTAR needs to cover most of the important weights with a particular block shape, it inevitably preserves many unimportant weights at the same time. Therefore, the compression ratio of this method is lower than that of the proposed method. In addition, the proposed method has a higher inference accuracy compared with that of LODESTAR. The column combining method combines complementary columns into one column after unstructured pruning to reduce the number of weights and alleviate the underutilization of systolic arrays. However, there are not always sufficient complementary columns to combine in a model. This is the reason why the compression ratio of our method is higher than that of the column combining method.

TABLE II
ACCURACY AND COMPRESSION RATIO WITH DIFFERENT METHODS

Dataset CIFAR-10	CifarNet		VGG-19	
	[10]	*Proposed*	*[11]*	*Proposed*
Accuracy (%)	62.52	63.23	94.7	93.6
Compression Ratio (%)	80.0*	87.15*	84.0	94.4
# Multiplexers	-	-	4096	128

* For weights in convolutional layers.

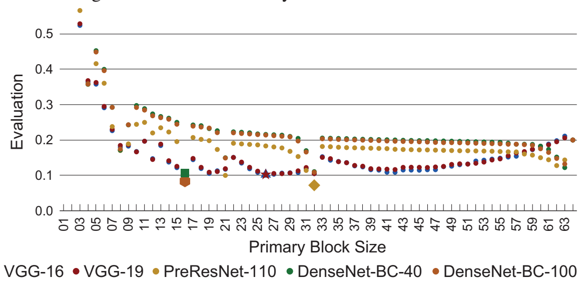

Fig. 6. Evaluation results for CNN models. The optimal block sizes are marked with different shapes.

In addition, the column-combining method requires a much larger number of multiplexers inserted into systolic flow to match weights and their corresponding inputs compared with the proposed method.

To select the optimal blocks to cover the weight matrix, we evaluate the latency and hardware cost for each combination of blocks. Fig. 6 shows the evaluation results of all block combinations for each CNN model. Since we prefer low hardware cost, the coefficients C_l, C_a, C_w here were set as 0.2, 0.75, 0.75, respectively. When the block size is small, the value of evaluation is large which indicates that the hardware cannot afford too many multiplexers. If the block size is too big, a large number of unimportant weights will be recovered which increases the total latency. We select the block size that represents the best tradeoff between latency and hardware cost minimization. For example, for DenseNet-BC-40 and DenseNet-BC-100, the optimal block size is 16×1. For VGG-16, the optimal combination of block sizes is $\{26 \times 1, 12 \times 1\}$.

To support different CNN models on systolic arrays, GA is implemented to search the locations of multiplexers to achieve a flexible systolic array structure. The initial systolic array is with *Row* 16, 26, 32, 48, 52 full of multiplexers, while the optimal sets of blocks for VGG-16, VGG-19, PreResNet-100, DenseNet-BC-40, and DenseNet-BC-100 are $\{26 \times 1, 12 \times 1\}$, 32×1, 16×1. Fig. 7 shows the flexible systolic array structure after optimization using GA. The line in Fig. 7 represents a set of continuous dots, and each dot denotes that there is an 8-bit multiplexer inserted in the PE. At the end of GA optimization, a certain quantity of multiplexers is removed without increasing the latency significantly. According to the locations of multiplexers, a flexible systolic structure is proposed with low latency and low hardware cost for various CNNs. For efficient support of VGG-16, VGG-19, PreResNet-100, DenseNet-BC-40, and DenseNet-BC-100 on the systolic array, the number of multiplexers in the final structure is 167, which is reduced by 47.81% compared with the initial design of five rows full of multiplexers. And 95.92% of multiplexers are reduced in comparison with the fully configurable design whose PEs are

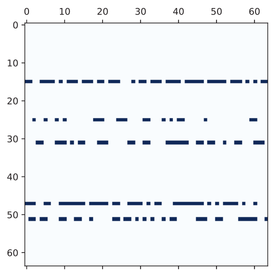

Fig. 7. Flexible systolic array structure after genetic algorithm optimization.

all equipped with multiplexers.

V. Conclusion

To exploit systolic arrays for the computations of various CNNs efficiently, a hardware-software codesign framework is proposed. By rearranging weight matrices, selecting suitable blocks, and multiplexing systolic array, a flexible systolic array structure is developed with accordingly pruned CNN models. The experimental results show that the latency can be reduced significantly compared with that of unstructured pruning with low hardware cost, while guaranteeing a high inference accuracy.

Acknowledgment

This work was supported in part by the National Key R&D Program of China under Grant 2018YFE0202800, and National Natural Science Foundation of China under Grant 61634004 and 61934002, and the Natural Science Foundation of Shaanxi Province for Distinguished Young Scholars under Grant no 2020JC-26, and the Youth Innovation Team of Shaanxi Universities.

References

[1] N. P. Jouppi *et al.*, “In-datacenter performance analysis of a tensor processing unit,” in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA, 2017, p. 1–12.

[2] B. Asgari, R. Hadidi, H. Kim, and S. Yalamanchili, “Eridanus: Efficiently running inference of dnns using systolic arrays,” *IEEE Micro*, vol. 39, no. 5, pp. 46–54, 2019.

[3] B. Asgari, R. Hadidi, and H. Kim, “Proposing a fast and scalable systolic array for matrix multiplication,” in *IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2020, pp. 204–204.

[4] Y. Wang, Y. Wang, H. Li, C. Shi, and X. Li, “Systolic cube: A spatial 3d cnn accelerator architecture for low power video analysis,” in *56th ACM/IEEE Design Automation Conference (DAC)*, 2019, pp. 1–6.

[5] S. Han, H. Mao, and W. J. Dally, “Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding,” *arXiv e-prints*, p. arXiv:1510.00149, 2015.

[6] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li, “Learning structured sparsity in deep neural networks,” in *Advances in Neural Information Processing Systems 29*, 2016, pp. 2074–2082.

[7] Z. Liu, J. Li, Z. Shen, G. Huang, S. Yan, and C. Zhang, “Learning efficient convolutional networks through network slimming,” in *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, 2017.

[8] J. Yu, A. Lukefahr, D. Palframan, G. Dasika, R. Das, and S. Mahlke, “Scalpel: Customizing dnn pruning to the underlying hardware parallelism,” in *ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017, pp. 548–560.

[9] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf, “Pruning Filters for Efficient ConvNets,” *arXiv e-prints*, p. arXiv:1608.08710, 2016.

[10] B. Asgari, R. Hadidi, H. Kim, and S. Yalamanchili, “Lodestar: Creating locally-dense cnns for efficient inference on systolic arrays,” in *Proceedings of the 56th Annual Design Automation Conference*, 2019.

[11] H. Kung, B. McDanel, and S. Q. Zhang, “Packing sparse convolutional neural networks for efficient systolic array implementations: Column combining under joint optimization,” in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, p. 821–834.

[12] A. Paszke *et al.*, “Pytorch: An imperative style, high-performance deep learning library,” in *Advances in Neural Information Processing Systems 32*, 2019, pp. 8026–8037.

[13] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, “cuDNN: Efficient Primitives for Deep Learning,” *arXiv e-prints*, p. arXiv:1410.0759, 2014.