MLCNN: Cross-Layer Cooperative Optimization and Accelerator Architecture for Speeding Up Deep Learning Applications

Beilei Jiang[1], Xianwei Cheng[1], Sihai Tang[1], Xu Ma[1], Zhaochen Gu[1], Song Fu[1],
Qing Yang[1], Mingxiong Liu[2]
[1]University of North Texas, [2]Los Alamos National Laboratory,
{beileijiang, xianweicheng, sihaitang, xuma, zhaochengu}@my.unt.edu;
{qing.yang, song.fu}@unt.edu; mliu@lanl.gov

***Abstract*—The ever-increasing number of layers, millions of parameters, and large data volume make deep learning workloads resource-intensive and power-hungry. In this paper, we develop a convolutional neural network (CNN) acceleration framework, named *MLCNN*, which explores algorithm-hardware co-design to achieve *cross-layer cooperative optimization and acceleration*. MLCNN dramatically reduces computation and on-off chip communication, improving CNN's performance. To achieve this, MLCNN reorders the position of nonlinear activation layers and pooling layers, which we prove results in a negligible accuracy loss; then the convolutional layer and pooling layer are co-optimized by means of *redundant multiplication elimination*, *local addition reuse*, and *global addition reuse*. To the best of our knowledge, MLCNN is the first of its kind that incorporates cooperative optimization across convolutional, activation, and pooling layers. We further customize the MLCNN accelerator to take full advantage of cross-layer CNN optimization to reduce both computation and on-off chip communication. Our analysis shows that MLCNN can significantly reduce (up to 98%) multiplications and additions. We have implemented a prototype of MLCNN and evaluated its performance on several widely used CNN models using both an accelerator-level cycle and energy model and RTL implementation. Experimental results show that MLCNN achieves $3.2\times$ speedup and $2.9\times$ energy efficiency compared with dense CNNs. MLCNN's optimization methods are orthogonal to other CNN acceleration techniques, such as quantization and pruning. Combined with quantization, our quantized MLCNN gains a $12.8\times$ speedup and $11.3\times$ energy efficiency compared with DCNN.**

***Index Terms*—Deep learning, Cross-layer optimization, Accelerators, Performance evaluation.**

I. INTRODUCTION

Convolutional neural network (CNN) has seen dramatic development recently, leading to increasing interests from industry, academia, and popular culture. However, the massive multiplication and accumulation (MAC) operations and frequent on-off chip data communications in CNN significantly affect its performance and wider adoption [1]. Accelerators, such as GPU, FPGA, TPU, and ASIC [2]–[4], have been developed to speed up MAC operations.

However, the on-off chip data communication consumes a significant amount of energy even with accelerators being used. As a result, recent research efforts have been focusing on optimizing both computation and communication, such as tiling and unrolling-based CNN execution, weight repetition-based CNN accelerators, and low-precision representations [5]. However, those techniques target the convolutional layer only without considering the interplay between multiple layers.

In this paper, we present a cross-layer cooperative algorithm-hardware co-design CNN optimization and acceleration framework, named MLCNN. Our design is based on a key observation that the relative order of activation and pooling does not affect CNN's accuracy. By reordering the two layers and co-optimizing the convolutional layer and pooling layer, we eliminate a large number of redundant multiplications and reuse many addition results. Moreover, we design and implement an MLCNN accelerator that targets the acceleration of CNN inference to accelerate the optimized CNN models.

The main contributions of this paper are as follows.

- We prove that CNN is insensitive to the relative order of activation and pooling, which enables cross-layer cooperative optimization.
- We design effective methods to identify redundant multiplications and local and global addition reuses in CNN. The proposed cross-layer optimization algorithm significantly improves the performance of CNN.
- We design and implement a prototype MLCNN at the register-transfer level (RTL) and evaluate its performance using an accelerator-level cycle and energy model. Experimental results on DenseNet, VGG, GoogLeNet and LeNet are very promising, i.e., MLCNN achieves a $3.2\times$ speedup and $2.9\times$ power consumption reduction compared with the dense CNN models (i.e., the original CNNs without using MLCNN optimization). Moreover, MLCNN can significantly improve the performance of other CNN acceleration techniques. Combining MLCNN with quantization achieves a $12.8\times$ speedup and $11.3\times$ power consumption reduction compared with dense CNNs.

The rest of the paper is organized as follows. Section II describes the key components in CNN networks. Section III proves the feasibility of reordering activation and pooling. Our cross-layer cooperative CNN optimization framework and algorithm are detailed and analyzed in Sections IV and V. Section VI describes our MLCNN accelerator and Section VII evaluates the performance of MLCNN. The related research is discussed in Section VIII. Section IX concludes the paper.

II. CONVOLUTIONAL NEURAL NETWORKS

A. CNN Architectures and Key Layers

A CNN model is composed of multiple functional layers, such as convolutional layers, activation layers, pooling layers,

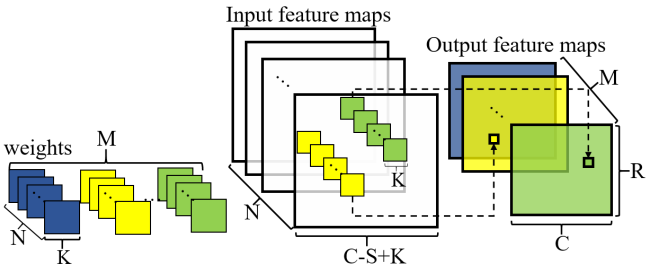

Fig. 1. The structure of convolutional layer in CNN.

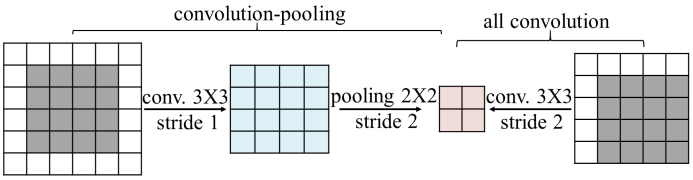

Fig. 2. Comparison of the spatial dimension reduction by convolutional-pooling CNNs and all convolutional CNNs. White pixels represent padding areas.

and fully connected layers. Among these layers, convolutional layers are the most computation-intensive.

The structure of a convolutional layer is depicted in Figure 1. N input feature maps with dimension $(\mathbf{S} \times \mathbf{C} + \mathbf{K} - 1) \times (\mathbf{S} \times \mathbf{R} + \mathbf{K} - 1)$ are dynamically juxtaposed against $\mathbf{M}$ $\mathbf{N} \times \mathbf{K} \times \mathbf{K}$ weight kernels. $\mathbf{M}$ three-dimensional $\mathbf{C} \times \mathbf{R}$ feature maps are generated. $\mathbf{S}$ represents the stride size used by a filter sliding the input feature maps. $\mathbf{N}$ and $\mathbf{M}$ denote the number of input and output channels respectively.

An activation function in CNN is non-linear. For example, ReLU and Sigmoid are two widely used activation functions. A standard ReLU is expressed by $y_{to,m,n} = max\left(C_{to,m,n}, 0\right)$. That is ReLU preserves the positive features while suppressing negative values to zero. Unlike ReLU, Sigmoid rescales a feature to (0, 1) with $y_{to,m,n} = \frac{1}{1+e^{-C_{to,m,n}}}$, where $y_{to,m,n}$ denotes the output of activation and $C_{to,m,n}$ is the output from the convolutional layer. Pooling is usually incorporated to extract new and representative information and discard useless or disruptive details [6]. Average pooling and max pooling are commonly used. The former employs an averaging operation (i.e., $f_{avg}(x) = \frac{1}{n}\sum_{i=1}^{n} x_i$) to smooth a feature map, while the latter applies a maximum operation (i.e., $f_{max}(x) = max\left(x_1, x_2, \ldots, x_n\right)$) to retain the maximum value among inputs while discarding the rest.

B. All Convolutional CNN

All convolutional (All-Conv) CNN is a special type of CNN contains convolutional layers only [7]. In All-Conv, pooling layers are replaced by increasing the stride of the preceding convolutional layers. Figure 2 compares the dimension variation of the conventional CNN and the All-Conv CNN. A stride-2 convolutional layer achieves a similar spatial dimension reduction as a convolutional layer followed by a stride-2 pooling. In comparison, a stride-2 convolutional layer achieves better performance, as it eliminates some floating-point multiplications and additions. However, pooling not only contributes to dimension reduction but also alleviates the sensitivity of outputs to shifts and distortions [6]. The exclusion of pooling layers loses these benefits and causes degradation of accuracy and robustness for CNNs. We present the accuracy results in Section III.

In this paper, we aim to accelerate CNN's performance and maintain the benefits brought from pooling. Moreover, we explore pooling to further reduce unnecessary multiplications and additions together with convolutional layers. The detail of our cross-layer CNN optimization method is described in Section IV.

III. Accuracy-Preserving CNN Layers Reordering

A. Reordering Activation and Pooling with Sensitivity Analysis

In many CNN networks, convolutional layers are followed by activation. The non-linear nature of activation makes the optimization of convolutional layers with other layers difficult. Meanwhile, pooling compresses and extracts representative features from the output of activation. It has been proved that ReLU followed by max-pooling behaves the same as max-pooling followed by ReLU [8]. In addition to max pooling, average pooling is widely used. The influence of switching activation and average pooling, however, has not been studied.

In this section, we analyze the sensitivity of model accuracy to the placement of average pooling in CNN networks. We reorder ReLU and average pooling layers in several widely used CNNs and study the all-conv counterparts. We use VGG16 [9], VGG19 [10], GoogLeNet [11] and LeNet5 [6] to illustrate and as case studies. Results on DenseNet [12] are presented in Section VII. The CNN models are trained using the CIFAR-10 and CIFAR-100 datasets [13]. Table I lists the number of convolutional layers and learnable parameters of the CNN models. The table shows the bigger a network (i.e., from LeNet5 to GoogLeNet) is, the more convolution operations and parameters are used.

TABLE I
Number of Convolutional Layers and Learnable Parameters in Studied CNN Models.

CNN models	# of Convolutional Layers	# of Learnable Parameters
LeNet5	1+1+1	62K
VGG16	2+2+3+3+3	14728K
VGG19	2+2+4+4+4	20040K
GoogLeNet	1+1+1+6+6+6+6+6+6+6+6	6166250K

After reordering the ReLU activation and average pooling layers (i.e., average pooling followed by ReLU), we measure the influence on models' accuracy. Figure 3 presents the top-1 and top-5 accuracy of the reordered CNN models compared with those of the original ones, and All-Conv [7] which replaces pooling by the stride-2 convolutional layer. The above accuracy is measured on the test datasets from CIFAR-10 and CIFAR-100.

In Figure 3, the reordered, original, and All-Conv CNN models exhibit similar top-1 and top-5 accuracy on CIFAR-10.

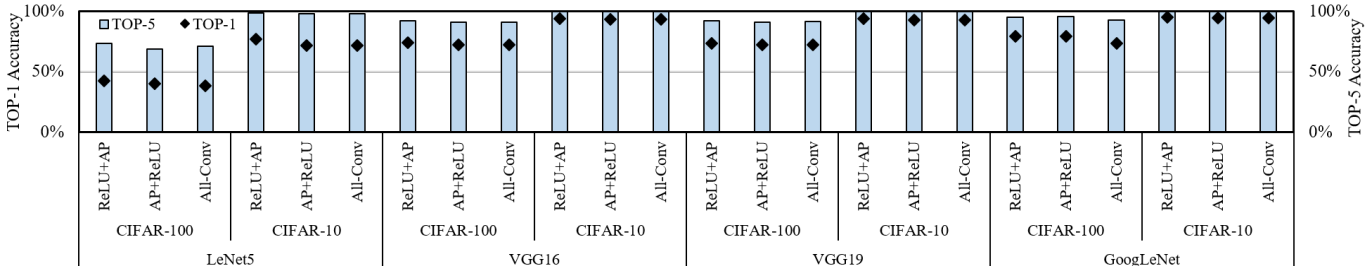

Fig. 3. Influence of reordering activation and pooling layers on CNNs' accuracy using CIFAR-100 and CIFAR-10 datasets. ReLU+AP: original CNN; AP+ReLU: reordered CNN; All-Conv: all convolutional CNN.

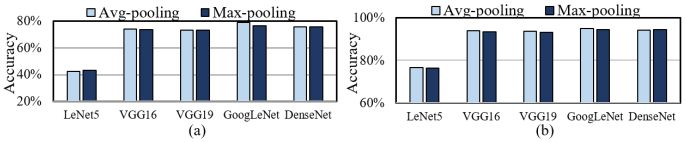

Fig. 4. Analyzing the influence of different pooling functions on the accuracy of CNNs using datasets (a) CIFAR-100 and (b) CIFAR-10.

Only LeNet-5 exhibits a little more top-1 accuracy degradation after reordering. The more complex CNN networks are, the better error tolerance they possess. Moreover, the figure shows the importance of average pooling for dealing with complex tasks on CIFAR-100. We can see the reordered network is superior to All-Conv, achieving higher top-1 and top-5 accuracy. Compared with the original CNN (ReLU-AP), the reordered GoogLeNet achieves about 0.5% improvement of the top-1 accuracy. A similar performance gain is observed in reordered VGG16 and VGG19. The more complex a CNN model is, the less sensitive the reordered CNN is in terms of model accuracy. These results motivate us to explore CNN acceleration by means of layer reordering. We also find that some deep CNNs, such as DenseNet and PNASNet [12], [14], already use a reordered CNN architecture, which further proves the feasibility of switching average pooling and ReLU. Overall, a marginal accuracy variation is observed by moving ReLU after average pooling, and pooling plays a critical role when handling complex tasks.

B. Average Pooling and Max Pooling

Pooling helps reduce a model's size and reduce over-fitting for CNNs. The main advantage of average pooling over max pooling is that average pooling does not remove information from input feature maps, while max-pooling keeps only the most distinct features and ignores the rest. We note some CNNs use max pooling, such as [15], [16]. We study the influence of different pooling functions (average and max) on CNNs' accuracy. Figure 4 plots the results. We can see average pooling outperforms max-pooling in most CNNs, which indicates it is beneficial to replace max-pooling with average pooling. Our MLCNN exploits average pooling to preserve useful information from feature maps.

IV. Cross-Layer CNN Optimization

MLCNN incorporates multi-layer cooperation and cross-layer optimization to effectively speed up CNNs. Specifically, we explore *redundant multiplication elimination* (RME), *local addition reuse* (LAR), and *global addition reuse* (GAR) across multiple layers to reduce expensive floating-point operations. To ease our discussion, we use the following symbols: $\mathbf{K}$ and $\mathbf{D}$ denote filter's and input's spatial dimensions, respectively, $\mathbf{S}$ is the step size, $\mathbf{I}$ represents an input feature map, $\mathbf{W}$ refers to the weight filter, $\mathbf{P}$ is the pooling output, and $\mathbf{N}$ is the number of elements in a row of a pooling feature map. Subscripts to, ti, x, and y denote indexes.

Layer-Reordered CNNs. As discussed in Section III, the relative order of activation and pooling has a marginal effect on the accuracy of a CNN model. By reordering these two layers, we generate an equivalent CNN network with convolutional, pooling, and activation layers. The reordered, equivalent network is used for cross-layer optimization.

Figure 5(a) presents an illustrating example. To facilitate discussion, we only present calculations of the first output feature from pooling $\mathbf{P}_{00}$. $\mathbf{P}$ denotes average pooling (AP) which is connected to a 2×2 neighborhood in the corresponding output feature map. The 2×2 neighborhood, i.e., $\mathbf{C}_{00}$ (-0.11), $\mathbf{C}_{01}$ (-0.12), $\mathbf{C}_{10}$ (-0.15), and $\mathbf{C}_{11}$ (-0.13), is averaged. Multiplication and accumulation (MAC) (denoted by M and A in the figure) calculates a convolutional output feature $\mathbf{C}_{x,y}$. In this example, a 2×2 weight filter slides across a 5×5 input feature map from the top left corner with a unit stride. In the beginning, the 2×2 weight filter covers the upper left area of the input, and after a MAC operation, $\mathbf{C}_{00}$ is produced. Then, the weight filter moves right by one column and starts a new MAC operation to calculate $\mathbf{C}_{01}$. Sliding downwards by one row followed by a MAC operation produces $\mathbf{C}_{10}$. After calculating $\mathbf{C}_{10}$, the weight filter slides right by one row followed by a MAC operation to determine $\mathbf{C}_{11}$. The output from MAC is adjusted by adding a bias. In total, 16 multiplications and additions are performed to process the input to pooling.

Identifying Redundant Multiplications and RME Optimization. After reordering average pooling to follow a convolutional layer, we examine MAC operations to find out whether some can be eliminated. As the weights used to calculate convolutional output features are the same, they can be fac-

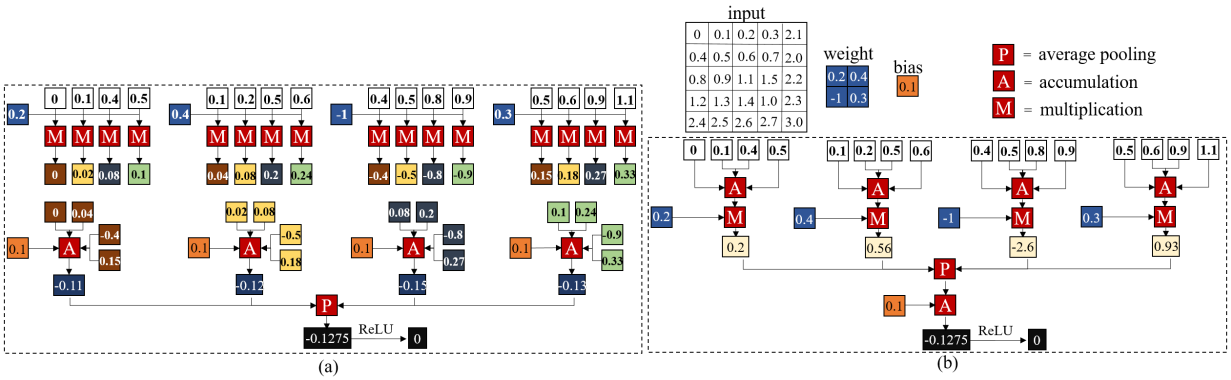

Fig. 5. An example illustrating the identification of redundant multiplications. A 5X5 input feature map is processed by a 2X2 filter. (a) Operations in the original CNN: pooling for generating feature P_{00}. (b) Weight factorization in calculating P_{00}.

tored out. After factorization, the multiplication of input and weight can be replaced by the accumulation of the inputs first. This can help eliminate many multiplications from the original CNN. As shown in Figure 5(b), four multiplications are used to calculate a pooling output feature after weight factorization, whereas in the original CNN, 16 multiplications are involved as shown in Figure 5(a). That is 75% of multiplications can be eliminated. We find that the number of multiplications saved in MLCNN is proportional to the pooling's filter size. Specifically, $\frac{\mathbf{K-1}}{\mathbf{K}}$ percent of multiplications can be eliminated, where $\mathbf{K}$ denotes the pooling's filter size. The value of $\mathbf{P}_{00}$ is the same, and thus the functional correctness of CNN is preserved.

Identifying Addition Reuses and LAR-GAR Optimization. After removing redundant multiplications, the CNN network still contains many additions. Next, we optimize those additions across layers to further improve performance and energy efficiency.

1) *Local Addition Reuse (LAR)*. In the fused convolutional-pooling layers, we find that the additions for adjacent input features in the same row or column are performed more than once and their results can be reused. We refer to the additions whose results can be shared in calculating a pooling output feature as *local addition reuse*. For example, in Figure 6(a), the additions between two inputs in the same column, i.e., $I_{01}+I_{11}, I_{02}+I_{12}, ..., I_{44}+I_{54}$, can be reused. Row-based LAR works in a similar way except that it retains the reused results from two elements in the same row.

To calculate a pooling output feature, $\mathbf{K^2}$ small accumulations (denoted by "A" in a square in Figure 6) and one major accumulation (denoted by "A" in a circle) are performed with 3 additions for each small accumulation and $\mathbf{K^2-1}$ additions for the major accumulation (without bias addition). In the original convolutional layer, there are $\mathbf{4\times K^2-1}$ additions in total. After applying local addition reuses, only $\mathbf{K\times(2\times K+S)+K^2-1}$ additions remain. The addition reduction rate is

$$P=\frac{K\times(K-S)}{4\times K^2-1}. \quad (1)$$

Algorithm 1: Cross-layer cooperative CNN optimization algorithm.

Input : I=input feature map; W=weight filter;
Output : O=output feature map;
Parameters : N=# of channels for I; M=# of channels for O; R'=# of rows for I; C'=# of columns for I; R=# of rows for O; C=# of columns for O; K= # of rows/columns for W;

```
for t_i = 0 : N-1 do
    for i = 0 : R'-1 do
        for j = 0 : C'-1 do
            HA_{t_i,i,j} = I_{t_i,i,j} + I_{t_i,i+1,j}; // half addition
        end
    end
end
for t_o = 0 : M-1 do
    for t_i = 0 : N-1 do
        for r = 0 : K-2 do
            for c = 0 : C'-1 do
                FA_{t_i,r,c} = HA_{t_i,r,c} + HA_{t_i,r,c+1}; // full addition
            end
        end
        for r = 0 : R-1 do
            for c = 0: K-2 do
                FA_{t_i,K+r-1,c} = HA_{t_i,K+r-1,c} + HA_{t_i,K+r-1,c+1};
            end
            for c = 0 : C-1 do
                FA_{t_i,K+r-1,K+c-1} = HA_{t_i,K+r-1,K+c-1} + HA_{t_i,K+r-1,K+c};
                for i = 0 : K-1 do
                    for j = 0 : K-1 do
                        O_{to,r,c} += W_{to,ti,i,j} × FA_{ti,i+r,j+c};
                    end
                end
            end
        end
    end
end
```

2) *Global Addition Reuse (GAR)*. When calculating different pooling output features, additions on the same portion of input features/pixels may be performed multiple times. We refer to those cases where addition results can be shared in calculating multiple pooling output features as *global addition reuses*. Figure 6(b) shows an example of row-based GAR. The "row" here refers to a row of the output feature map from pooling instead of the activation feature map used in the LAR design. In Figure 6, the addition results from calculating $\mathbf{P}_{00}$, i.e.,

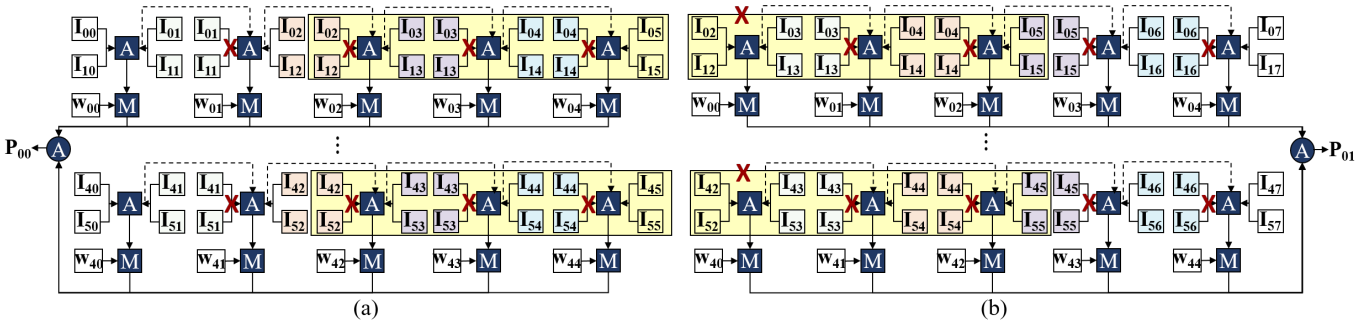

Fig. 6. Local addition reuses (LARs) and global addition reuses (GARs). The cross marks in (a) show LARs and the highlighted blocks in (b) show GARs.

$I_{02}+I_{12}+I_{03}+I_{13}, I_{03}+I_{13}+I_{04}+I_{14}, I_{04}+I_{14}+I_{05}+I_{15}, ..., I_{44}+I_{54}+I_{45}+I_{55}$, can be reused to calculate $\mathbf{P}_{01}$. We use a row in the output matrix from pooling as the unit to illustrate GAR and calculate the addition reduction rate. There are also column-based GARs, which refer to the additions that are shared in calculating the output features from pooling in the same column. More computations can be eliminated by combining the row-based and column-based GARs.

Recall that $\mathbf{3 \times K^2 + K^2 - 1}$ additions are used to calculate an output feature from pooling. $\mathbf{N \times (3 \times K^2 + K^2 - 1)}$ additions are performed to produce a row of output features. GAR becomes more beneficial when small accumulations than major accumulations are used. The reusable small accumulations get results directly from the previous additions. Consequently, only $\mathbf{K \times (D - S)}$ small accumulations are left. Since there are 3 additions in each small accumulation, the number of additions can be reduced to $\mathbf{K \times (D - S) \times 3}$. Taking the major accumulations into account, there are $\mathbf{K \times (D - S) \times 3 + N \times (K^2 - 1)}$ additions in total. The addition reduction rate attributed to GARs is

$$P = \frac{3 \times N \times K^2 - 3 \times K \times (D - S)}{N \times (3 \times K^2 + K^2 - 1)}. \quad (2)$$

Cross-Layer Cooperative CNN Optimization Algorithm. We design a cross-layer cooperative optimization algorithm to explore redundant multiplication elimination (RME), local addition reuse (LAR), and global addition reuse (GAR) to speed up deep learning computation. Algorithm 1 presents the pseudo-code. The convolutional-pooling layers run in three major steps: half addition, full addition, and MAC. A half addition adds two adjacent elements in the same column. A full addition exploits LAR to calculate the accumulation result, i.e., I_Acc in Equation (3). GAR is used by MAC to produce the final output.

$$P_{to,x,y} = Relu\left(\frac{\sum_{i=0}^{K-1}\sum_{j=0}^{K-1}\sum_{ti=0}^{N-1} w_{to,ti,i,j} \times I\_Acc}{4.0} + B_{to}\right). \quad (3)$$

V. Performance Analysis of MLCNN

As discussed in Section IV, redundant multiplication elimination (RME) can reduce $\frac{\mathbf{K-1}}{\mathbf{K}}$ (percentage) of multiplications in convolutional-pooling layers, where K denotes the filter size in pooling layers. The performance benefit from addition reuses (i.e., LAR and GAR) is influenced by several factors, including filter size, step size, and input dimension. In this section, we analyze the performance gain brought by LAR and GAR. The following symbols are used in our discussion: $\mathbf{K}$ (the spatial dimension of a filter), $\mathbf{S}$ (step size), $\mathbf{D}$ (the dimension of an input feature map), and $\mathbf{N}$ (the number of elements in a row of a pooling feature map).

TABLE II
Impact of filter size on local addition reuses (LAR) in calculating pooling output features. A unit stride is used.

Filter size	#of additions w/o LAR	#of additions w/ LAR	Addition reduction rate (%)
11 × 11	483	373	22.8
9 × 9	323	251	22.3
7 × 7	195	153	21.5
5 × 5	99	79	20.2
3 × 3	35	29	17.1
2 × 2	15	13	13.3

Local addition reuses: The influence of filter size and step size on the effectiveness of addition reuses is presented in Tables II and III, respectively. In Table II, we can see the number of additions reduced by LAR closely depending on the filter size. The lowest reduction rate occurs when a 2×2 filter is used. A larger filter leads to more additions reused.

Does a larger filter always result in more addition reuses? To answer this question, we formally analyze the relation between filter size and addition reuses. The addition reduction rate P under a unit stride can be determined as follows.

$$P = \frac{K \times (K - 1)}{4 \times K^2 - 1}. \quad (4)$$

Equation (4) shows a positive linear relation between P and the filter size at the beginning. The rate flats as the filter size increase further with P approaching 25%.

Then, we fix the filter size (e.g., 11×11) and vary the step size. We choose 11×11 as it is commonly used in CNN models (e.g., AlexNet [17]) and it leads to the highest addition reduction rate (shown in Table II). The influence of step size on addition reuses is presented in Table III. From the table, we find the addition reduction rate decreases almost linearly as the step size increases. The greatest addition reduction is achieved when a unit step size is used, and the performance

TABLE III
IMPACT OF STEP SIZE ON LOCAL ADDITION REUSES (LAR) IN CALCULATING POOLING OUTPUT FEATURES.

Step size	#of additions w/o LAR	#of additions w/ LAR	Addition reduction rate (%)
1	483	373	22.8
2	483	384	20.5
3	483	395	18.2
4	483	406	15.9
5	483	417	13.7
6	483	428	11.4
...	...	...	...
11	483	483	0

gain drops when the step size becomes greater than the filter size.

Global addition reuses: We also analyze the effectiveness of GAR. A unit stride and a 28×28 input feature map are studied. Table IV shows our analytical results. We observe the addition reduction rate increases and approaches the apex where a 15×15 filter is used. Then the effectiveness of addition reuses drops as the filter size goes up further.

TABLE IV
IMPACT OF FILTER SIZE ON GLOBAL ADDITION REUSES (GAR) IN CALCULATING A ROW OF POOLING OUTPUT. THE RESULTS ARE BASED ON A 28×28 INPUT FEATURE MAP AND A UNIT STRIDE.

Filter size	#of additions w/o GAR	#of additions w/ GAR	Addition reduction rate (%)
3×3	455	347	23.7
5×5	1188	693	41.7
13×13	5400	2397	55.6
15×15	6293	2783	55.8
17×17	6930	3105	55.2

The effectiveness of GAR is also affected by the step size. In Table V, we find the smallest step size leads to the highest global addition reuse rate (i.e., 55.6%). The effectiveness of GAR drops dramatically as the step size increases.

TABLE V
IMPACT OF STEP SIZE ON THE PERFORMANCE GAIN FROM GAR IN CALCULATING A ROW OF THE POOLING OUTPUT. THE RESULTS ARE BASED ON A 13×13 FILTER AND A 28×28 INPUT FEATURE MAP.

step size	#of additions w/o GAR	#of additions w/ GAR	Addition reduction rate (%)
1	5400	2397	55.6
3	2025	1479	27.0
5	1350	1233	8.7

Furthermore, we investigate the relationship between input dimension and the effectiveness of addition reuse. A 13×13 filter and a unit stride are used. Table VI lists the results. We observe a positive correlation between the input dimension and the addition reduction rate. We further formulate the relation between them. To calculate a row of the pooling output without GAR, $(4 \times K^2 - 1) \times (\frac{D-K+1}{2})$ additions are performed. When $K = 13$, the number of additions is $337.5 \times D - 4050$. When GAR is applied, the number drops to $(4 \times K^2 - 1) + ((6 \times K + K^2 - 1) \times (\frac{D-K+1}{2} - 1))$. It is $123 \times D - 1047$, when $K = 13$. Therefore, the addition reduction rate P equals

$$P = \frac{214.5 \times D - 3003}{337.5 \times D - 4050}. \quad (5)$$

As $\mathbf{D}$ increases, $\mathbf{P}$ approaches 63.6%.

$$\lim_{D \to \infty} \frac{214.5 \times D - 3003}{337.5 \times D - 4050} = 0.636. \quad (6)$$

TABLE VI
IMPACT OF INPUT DIMENSION ON THE PERFORMANCE GAIN FROM GAR IN CALCULATING A ROW OF THE POOLING OUTPUT.

input dim	#of additions w/o GAR	#of additions w/ GAR	Addition reduction rate (%)
28X28	5400	2397	55.6
32X32	6750	2889	57.2
224X224	71550	26505	63.0

When both GAR and LAR are applied, the additions in small accumulations can be reused, which causes the number of additions to drop from $(4 \times K^2 - 1)$ to $(K^2 - 1)$.

$$\lim_{K \to \infty} \frac{3 \times K^2}{4 \times K^2 - 1} = 0.75. \quad (7)$$

That is up to 75% of additions can be saved.

VI. MLCNN ACCELERATOR

We qualitatively demonstrate the effectiveness of MLCNN for CNN optimization. However, its potential cannot be fully exploited without an efficient computing platform that provides high-throughput data communication and computation. As such, we develop a domain-specific accelerator architecture to support MLCNN.

A. Architectural Design of MLCNN Accelerator

Deep learning applications need to process large columns of data. To achieve high-throughput dataflow, we design an addition reuse (AR) unit and exploit shift registers in MAC slices to perform LAR and GAR optimization. Addition reuses also help prevent unnecessary data movement. To reduce on-off chip data communications, our MLCNN accelerator adds two adjacent features in the same column and transfers the results (instead of the original data) to DRAM. We also develop reconfigurable AR units, MAC slices, and preprocessing functions to execute the fused convolutional-pooling layers and those convolutional layers which are not followed by pooling. In this section, we present the architecture and design of the MLCNN accelerator and then describe the dataflow design, followed by explaining how to leverage LAR and GAR, preprocessing, and reconfigurability of the accelerator to speed up CNN execution.

Architecture of the MLCNN Accelerator. The MLCNN accelerator is tailored to accelerate CNNs. Figure 7 shows its architecture and major components. The execution of the MLCNN accelerator is managed by a controller. Processing elements (PEs) are designed to perform 32-bit floating-point or 8-bit fixed-point multiplications. Accordingly, the addition unit

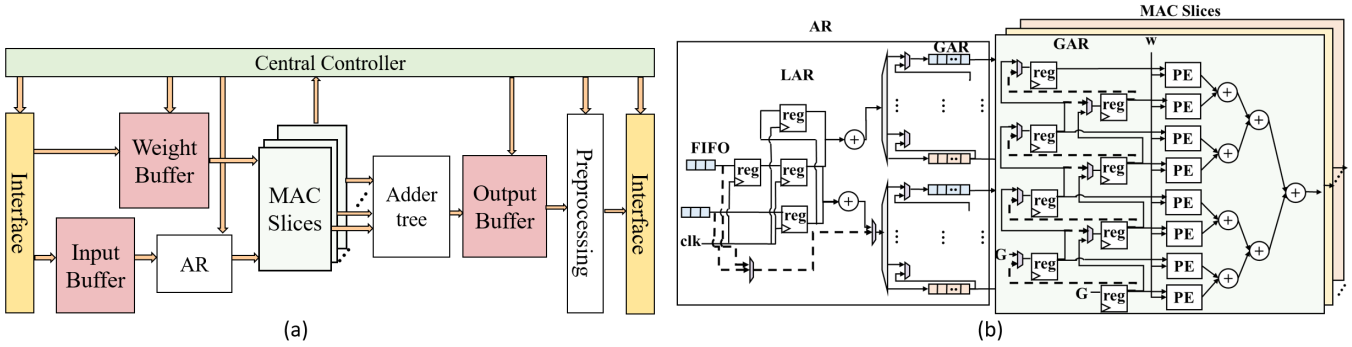

Fig. 7. MLCNN acceleration. (a) Overall architecture. (b) Block diagrams of the addition reuse (AR) unit and multiplication and accumulation (MAC) slices.

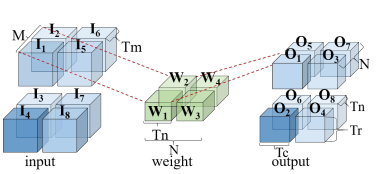

Fig. 8. Dataflow design for MLCNN accelerator.

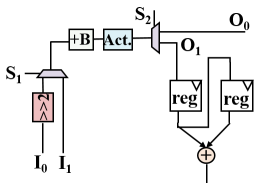

Fig. 9. Preprocessing.

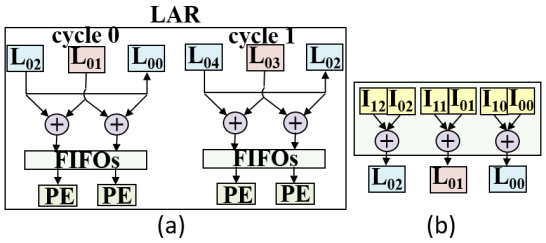

Fig. 10. (a) Leveraging LAR on the MLCNN accelerator. (b) Part of the preprocessing.

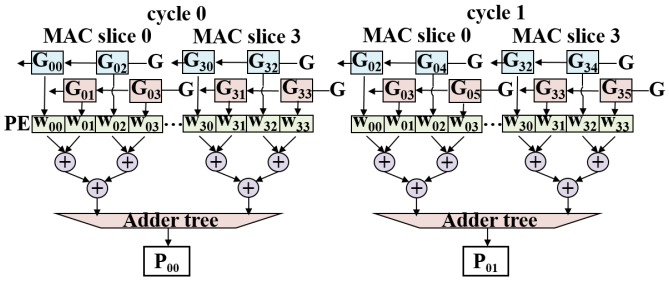

Fig. 11. Design of MAC slices for performing GAR.

adds two 32-bit floating-point or 8-bit fixed-point operands. LAR and GAR are performed by AR units and MAC slices. The input data and weights are stored in a *multi-bank input-weight buffer* which aims to hide the latency in the on-off chip data communication. The output buffer, on the other hand, is responsible for data accumulation in addition to storing results. For fused convolutional-pooling layers, two adjacent outputs in the same column are added in preprocessing before the result is sent to the off-chip DRAM.

Dataflow Design. A *weight-input reuse dataflow* is designed to take advantage of the extensive reuse of weight operands and the AR unit for input reuses. Weights are fetched from PE's registers and reused for different inputs. Weight is not replaced from a register until it has been multiplied by all inputs.

Moreover, we explore loop tiling to place a chunk of data in the buffer and leverage temporal locality to improve performance with a limited buffer space. Four major parameters are used in loop tiling for a convolutional layer, i.e., $< T_m, T_n, T_r, T_c >$ [18]. For an input feature map with M channels and an output feature map with N channels, the input and output feature maps are partitioned into $\frac{M}{T_m}$ and $\frac{N}{T_n}$ tiles, respectively. Accordingly, R rows and C columns in the output feature map are tiled into $\frac{R}{T_r}$ and $\frac{C}{T_c}$ chunks.

Figure 8 illustrates the dataflow with loop tiling. T_m, T_n, T_r and T_c are halves of the values of M, N, R, and C, respectively. Two tiles in different channels are processed consecutively in order to reduce the number of outputs temporarily buffered, e.g., $I_1 \rightarrow I_2$, $I_3 \rightarrow I_4$ shown in the figure. In the beginning, a weight chunk $\mathbf{w_1}$ is loaded into PE's registers, and data from the input chunk $\mathbf{I_1}$ is added in the AR unit and sent to the PE. $\mathbf{w_1}$ can be replaced in the register after it is multiplied with all the inputs in $\mathbf{I_1}$. This process repeats for the weight chunk $\mathbf{w_2}$ and input chunk $\mathbf{I_2}$. $\mathbf{I_3}$ is processed after $\mathbf{I_2}$ (instead of $\mathbf{I_4}$) is multiplied by $\mathbf{w_2}$. The preceding steps repeat until the output is produced.

AR Unit. An AR block consists of two addition units, four registers, two demultiplexers, several multiplexers, and FIFOs. Two small FIFOs, i.e., the leftmost ones shown in Figure 7(b), are used for LAR optimization. Two operands are sent to the AR unit every clock cycle at runtime. The operand that has an odd column index is stored in the small FIFO, that is the top one in the figure. The other operand is stored in the bottom FIFO, and then moved from the bottom register to the top one. One operand is shared by the two addition units, which corresponds to the column-based LAR. Figure 10 illustrates how LAR is conducted on the MLCNN accelerator, where $\mathbf{L}$ denotes the input addition result which comes from preprocessing. In the figure, we can see data sharing between registers and data reuses are achieved from using shift registers.

MAC Slices. The outputs of the two addition units in LAR are cached in the rightmost FIFOs before being fed to the

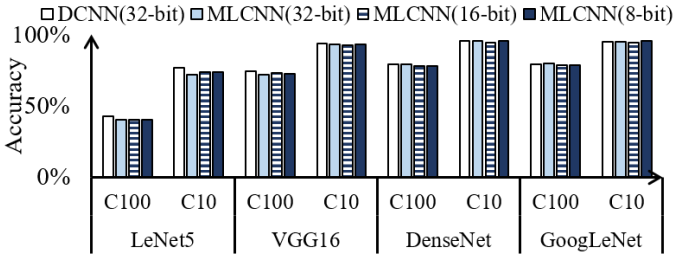

Fig. 12. Accuracy comparison between MLCNN and quantized MLCNN on CIFAR-100 (C100) and CIFAR-10 (C10) datasets.

MAC slices for GAR optimization. With two FIFOs in an MAC slice, the number of FIFOs is twice the number of MAC slices. When they are full, the top FIFOs will not accept data from the addition units until the computation for the next row in the output feature map is initiated. In the MAC slice, two shift register sets are used for GAR.

Figure 11 presents the data and control flow in a MAC slice. $\mathbf{G}$ denotes the accumulation result from the inputs, i.e., $G_{00} = I_{00} + I_{01} + I_{10} + I_{11}$. Weights and accumulation results are processed/computed in PE. To multiply 32-bit operands, we develop a PE with a three-stage pipeline [19]. We use a Wallace Tree multiplier to perform 8-bit fixed-point multiplications [20]. An adder tree is employed to produce the final output.

Preprocessing. Before being transferred to the off-chip DRAM, data is pre-processed as shown in Figure 9. We use a selector $\mathbf{S_1}$ to control the execution mode of a layer. When $\mathbf{S_1} = 0$, a fused convolutional-pooling layer is executed, while $\mathbf{S_1} = 1$ leads to the regular mode (i.e., original convolutional and pooling layers). When $\mathbf{S_1} = 0$, $\mathbf{I_0}$ is selected, otherwise $\mathbf{I_1}$ is used. $\mathbf{I_0}$ is divided by four by shifting right to generate the pooling result, as shown in Figure 9. It is adjusted by adding a bias, followed by passing through an activation function. The selector $\mathbf{S_2}$ is governed by the incoming layer. For a fused layer, $\mathbf{O_1}$ is selected and $\mathbf{S_2} = 1$, whereas a regular layer has $\mathbf{S_2} = 0$ (i.e., $\mathbf{O_0}$ dominates).

Reconfigurability. In Figure 7(b), the dashed lines show the execution of the traditional convolutional layer. This execution path is similar to those in state-of-the-art CNN accelerators. Input features are sent to PE without addition, followed by a weight-input multiplication. FIFOs are used to take advantage of CNN's input reuse opportunities. The solid lines show the execution of fused convolutional-pooling layers. Additional components (including registers, FIFOs, AR units, and MAC slices) are provided in our design to speed up RME, LAR and GAR optimizations. Preprocessing supports both the fused and regular execution modes. The selectors $\mathbf{S_1}$ and $\mathbf{S_2}$ determine which execution mode MLCNN is currently in.

VII. Performance Evaluation of MLCNN

We evaluate the performance of MLCNN in a PyTorch [21] environment using four representative CNNs, i.e., DenseNet, VGG16, GoogLeNet, and LeNet5 on CIFAR-100 [13]. Twelve layers in GoogLeNet and three layers in the transition blocks in DenseNet can benefit from MLCNN's optimization. VGG-16 has five convolutional layers that can be optimized with pooling layers. In LeNet-5, two convolutional layers benefit from MLCNN's optimization while others are not affected since LeNet-5 has two pooling layers.

TABLE VII
Configurations of accelerators in experiments.

	DCNN FP32	MLCNN FP32	MLCNN FP16	MLCNN INT8
#MAC Slices	32	32	64	128
Bitwidth	32	32	16	8
area (mm^2)	1.52	1.52	1.52	1.52
On-chip memory (kB)	134	134	134	134

We have implemented an MLCNN accelerator and synthesized it at the register-transfer level (RTL) written in Verilog. We use Design Compiler with the 45-nm TSMC library to analyze the area of MAC Slices under different designs. We use CACTI [22] to measure the power consumption and Xilinx Vivado [34] to measure the performance of our MLCNN accelerator. Table VII lists the configuration of our MLCNN accelerator employed in the experiments. MLCNN and baseline DCNN have the same amount of on-chip memory which caches the input and weights for reducing the off-chip memory access latency. For fair comparisons, we use the same number of MAC Slices and the same area budget ($1.52mm^2$) for MLCNN and the baseline DCNN. We evaluate the performance of MLCNN in terms of execution time, floating-point operation reduction, and energy efficiency.

A. Accuracy of MLCNN

In the first set of experiments, we aim to understand to what extent the design of MLCNN influences the accuracy of object classification compared to the original DCNN. As the optimization methods in MLCNN are complementary to the existing CNN acceleration techniques, we combine MLCNN with quantization (a widely used CNN acceleration approach) in addition to the standalone MLCNN.

We perform input/activation and weight quantization adapted from DoReFa-Net [23]. Both weight and input are quantized based on an extensive straight-through estimator (STE) method [24], expressed as

$$r_o = quantize_k(r_i) = \frac{1}{2^k - 1} round((2^k - 1) r_i), \quad (8)$$

where a real number r_i in [0, 1] is quantized to k-bit output r_o in [0, 1].

For weight quantization, as weights may be positive or negative, we use the following method.

$$r_o = 2quantize_k(\frac{tanh(r_i)}{2max(|tanh(r_i)|)} + \frac{1}{2}) - 1, \quad (9)$$

where tanh rescales the range of weights to [-1, 1] before being quantized to k-bit, and $\frac{tanh(r_i)}{2max(|tanh(r_i)|)}$ assures the value is in [0, 1] (the maximum operation is performed on all the weights

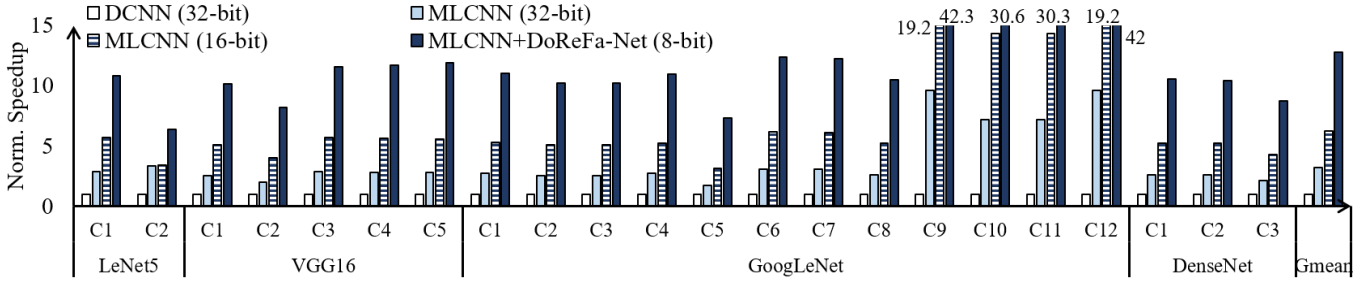

Fig. 13. Speedup of MLCNN (FP32, FP16 and INT8) compared with DCNNS: DenseNet, VGG16, GoogLeNet, and LeNet5. "C" denotes a convolutional layer.

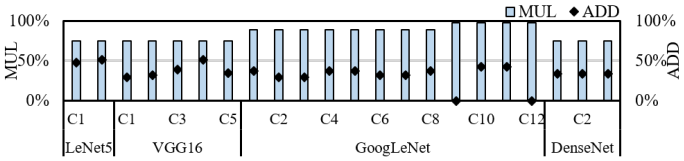

Fig. 14. Percentage of FLOPs reduced by MLCNN."C" denotes a convolutional layer in a CNN model.

of that layer). Function $quantize_k()$ quantizes its input to a k-bit fixed-point value in [0, 1]. An affine transform then scales the k-bit weight back to [-1, 1].

For input quantization, the extensive STE method, i.e., Equation (8), is used on input layers after ReLU, and the extended method, i.e., Equation (9), is used on the inputs without ReLU before them.

Figure 12 plots the measured accuracy of DCNN and MLCNN on four CNN models (DenseNet, VGG16, GoogLeNet, and LeNet5). A static quantization scheme, DoReFa-Net [23], is also used with MLCNN. As can be seen, although LeNet5 gives a 1.5% accuracy degradation, the MLCNN optimization on GoogLeNet results in around 0.5% accuracy elevation compared with DCNN. It is rational as complex CNNs will have more error tolerance. MLCNN and DCNN reach similar accuracy on CIFAR-10 and CIFAR-100 for FP32, FP16, and INT8. In comparison with MLCNN, the most accuracy degradation of quantized MLCNN (8-bit) comes with GoogLeNet on CIFAR-100, which is less than 0.8%. VGG16, however, has about a 0.5% accuracy elevation on CIFAR-100. Overall, MLCNN, DCNN, and quantized MLCNN (8-bit) are equivalent in terms of accuracy.

B. CNN Acceleration Performance by MLCNN

To quantify the performance improvement brought by MLCNN, we measure the execution time of CNNs. For fair comparisons, the same area budget ($1.52mm^2$) and on-chip memory is applied to the quantized MLCNN and the quantized DoReFaNet on 8-bit operands. Figure 13 presents the performance of MLCNN (FP32 and FP16) and quantized MLCNN (INT8) compared with the baseline DCNNs. The results show MLCNN achieves about 3.2× performance improvement on average for 32-bit floating-point operations. MLCNN FP16 achieves a 6.2× speedup compared with DCNNs. Moreover, the quantized MLCNN (INT8) has a 12.8× performance gain. Given the same area budget, more MAC slices are implemented under a lower precision as shown in Table VII. The forward propagation convolutional layer (C9) has the highest performance gain in GoogLeNet (9.63×, 19.2× and 42.3× for FP32 MLCNN, FP16 MLCNN and quantized MLCNN, respectively). Overall, GoogLeNet benefits the most performance improvement than other CNN models explored in this work. As mentioned in section IV, the percentage of multiplications saved is proportional to the pooling filter size. Therefore, it is rational that GoogLeNet achieves the most performance gain with the highest pooling filer size (8 × 8). The significant reduction of the execution time comes from the elimination and reuse of redundant floating-point operations including multiplications and additions.

C. Acceleration of Floating-Point Computations

Floating-point multiplications and additions are compute-intensive and power-hungry. We measure the number of floating-point operations saved by MLCNN in DenseNet, VGG-16, GoogLeNet, and LeNet-5. Figure 14 shows the results. There are two average pooling layers after the first two convolutional layers in LeNet-5. MLCNN optimizes the first and second convolutional layers. For VGG-16, five layers can be fused and cross-optimized, i.e., Layers 1 to 5 which are convolutional layers. Twelve convolutional layers in GoogLeNet and three in DenseNet benefit from MLCNN's optimization. From the figure, we can see RME contributes to up to 98% multiplication reduction. The percentage of multiplication eliminated is $\frac{\mathbf{K-1}}{\mathbf{K}}$, where K denotes the filter size in pooling. Furthermore, different layers exhibit varying addition reduction rates. Convolutional layer 2 in LeNet-5 shows the greatest addition reduction, 51.52%. A 1 × 1 filter together with a 1 × 1 output feature map disables addition reuse in the design of MLCNN. As a result, no addition is eliminated in such layers.

Overall, DenseNet, VGG16, and LeNet-5 save 75% of multiplications. For GoogLeNet, up to 98% of multiplications can be eliminated. LeNet5 exhibits the highest addition reduction rate, i.e., 51.52%, whereas no addition is reused in DenseNet. A small dimension filter used in DenseNet, VGG16, and GoogLeNet (i.e., 3 × 3 or 1 × 1) results in a relatively less addition reductions compared to LeNet5 (5 × 5 filter).

The performance benefit from addition reuses is influenced by the filter size, step size, and input dimension (See Section V). Figure 14 shows the average-case results using several

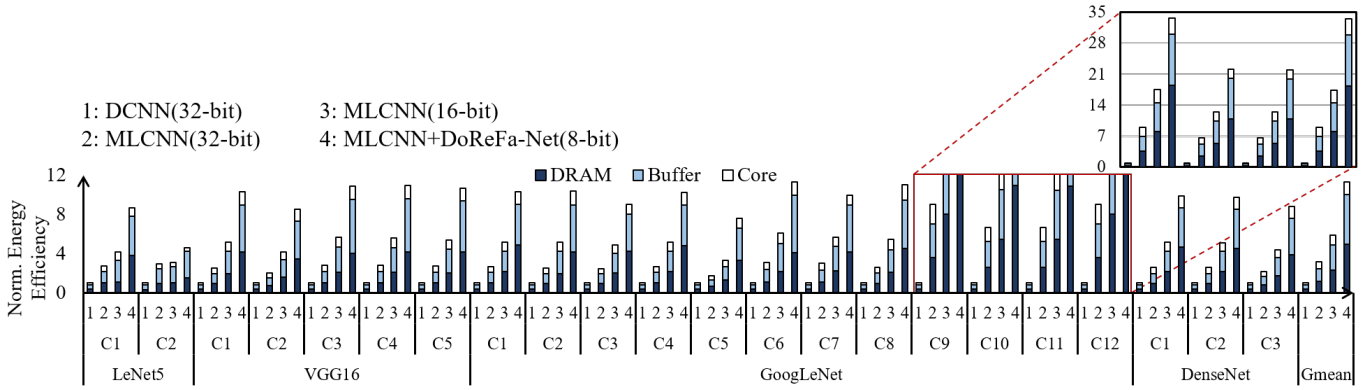

Fig. 15. Breakdown of energy consumption by MLCNN (FP32, FP16 and INT8) compared with DCNNS: DenseNet, VGG16, GoogLeNet, and LeNet5."C" denotes a convolutional layer.

real-world CNN models. In the best case, 75% of additions can be reduced by leveraging local addition reuses (LARs) and global addition reuses (GARs) when the filter dimension (**K**) approaches infinity in Equation (7). In a real implementation, on the other hand, **K** is usually a small number. As a result, the CNN models that we evaluate on CIFAR-100 achieve 51.5% or less addition reduction.

D. Energy Efficiency

Machine learning workloads are compute-intensive and power-hungry. We measure energy saving by using MLCNN. Figure 15 plots the energy consumption by MLCNN compared with DCNNs on FPGA measured by CACTI [22]. In the figure, we can see MLCNN consumes a less amount of energy under all precision modes (FP32 MLCNN, FP16 MLCNN and INT8 quantized MLCNN). On average, MLCNN achieves a $2.9\times$ energy efficiency on FP32 operations compared with DCNN. For FP16 MLCNN, a $5.9\times$ energy efficiency is obtained. Moreover, the quantized MLCNN (INT8) is $11.3\times$ energy-efficient than DCNNs. In particular, the Convolutional Layer #9 in GoogLeNet achieves the greatest energy efficiency, i.e., over $9.0\times$ by FP32 MLCNN, $17.5\times$ by FP16 MLCNN and $33.6\times$ by INT8 quantized MLCNN compared with DCNN, among all the layers.

To gain a deeper understanding of MLCNN's energy efficiency, we further study the energy consumption by the three major components, that is DRAM, Buffer (input, weight, and output buffer), and processing cores (MAC). As shown in Figure 15, all these components (i.e., DRAM, Buffer, and MAC) contribute to the energy reduction on FP32 MLCNN, FP16 MLCNN, and quantized (INT8) MLCNN. Specifically, DRAM, Buffer, and MAC contribute to reducing the execution time which dominates the static energy use. The dynamic energy is mainly saved by the Buffer and MAC. Data reuse by MLCNN eliminates redundant data accesses to the Buffer. Moreover, the number of multiplications and additions in MAC is significantly reduced by MLCNN.

VIII. RELATED WORK

CNNs are incorporating more layers to extract features and build accurate models, which slows down both model training and inference. To speed up CNNs, researchers have proposed optimization techniques and accelerators.

GPU, ASIC, and FPGA are the major CNN accelerators. GPU excels in parallel computing. However, the prohibitive power consumption makes GPU less preferable in terms of performance per watt. Custom ASICs, such as Tensor Processing Unit (TPU) [3], were developed. TPU is $15\times$-$30\times$ faster than GPU and CPU when running CNNs. In addition, NVIDIA developed tensor cores [25] that feature efficient multi-precision matrix multiplications for CNNs. FPGA-based accelerators attract new attention. For example, a tiling and unrolling-based CNN execution framework on FPGA was proposed, which balances on-chip computational resource and memory bandwidth [26]. Moreover, Alwani et al. presented a fused-layer CNN that fully utilizes the memory space to mitigate data transfer overhead [27]. They focused on data reuses between layers to reduce DRAM accesses, which did not speed up multiplication and addition operations in convolutional layers. Our MLCNN targets multiplication and addition optimization in convolutional layers. The experimental results show MLCNN (achieving a 3.2X speedup) is more effective than fused layers (i.e., 1.5X for the first 2 convolutional layers in AlexNet).

Among the software solutions, sparsity has been extensively studied [28]. Synapses pruning can result in $10\times$ data reduction, speeding up training and inference [29]. Dataflow optimization has been explored to accelerate CNN applications. For example, Eyeriss [30] provides a dataflow taxonomy and row-stationary dataflow. In [31], a heterogeneous dataflow accelerator is proposed for convolutional layers. Moreover, the error tolerance property of CNNs enables low-precision calculations. Han et al. [32] achieved a $35\times$ weight size reduction for AlexNet. Hegde et al. [33] proposed a weight repetition framework that explores the low-precision representations of weights for acceleration. MLCNN optimizes expensive multiplications and additions across multiple layers in CNNs, which is complementary to the preceding techniques.

Daultani et al. [8] reordered max pooling and ReLU, which could reduce some binary max operations. Specifically, they targeted the $max(a, b)$ function. It worked as follows. In a tra-

ditional CNN with convolution-ReLU-pooling layers, assume the dimension of a feature map output from the convolutional layer is 3×3. Then, ReLU carries out 9 binary max operations. With a 2×2 pooling filter, there are 12 binary max operations in the max-pooling layer. In total, 21 binary max operations are performed. After reordering max-pooling and ReLU, ReLU conducts 4 binary max operations and max-pooling still needs 12 binary max. The overall number of binary max operations is reduced from 21 to 16 (i.e., a 23.8% reduction). However, as binary max is a light-weight operation, which accounts for only 4% of execution time compared with over 90% of the time by run convolutional layers [35], its overall speedup was limited, that is around $1.03\times$ [8]. Floating-point multiplications and additions in convolutional layers involve heavy computation. However, they were untouched in [8].

In contrast, our MLCNN targets directly the compute-intensive convolutional layers by co-optimizing the convolutional layer and pooling layer. Specifically, we identify redundant multiplications and local and global addition reuses. Up to 75% of floating-point multiplications and additions are reduced and our performance improvement is significant, i.e., a $3.2\times$ speedup.

IX. Conclusions

In this paper, we present MLCNN which speeds up deep learning applications and develop an efficient CNN accelerator. We design a cross-layer cooperative optimization method to achieve redundant multiplication elimination, local addition reuse, and global addition reuse. Both statistical analysis and experimental results show MLCNN can significantly improve the performance and reduces power consumption for deep learning, which makes MLCNN a promising solution for high-performance and low-power deep learning applications and systems. We present the results on DenseNet which includes bypass connections between non-adjacent convolutional layers. MLCNN can also be applied to ResNet. The convolutional layers with pooling in ResNet-18 can benefit from MLCNN with layer reordering and cross-layer optimization.

In our future research, we plan to integrate MLCNN with the latest CNN networks and leverage our optimization method to speed up other deep learning technologies, such as graph neural networks and recurrent neural networks.

Acknowledgment

This work has been supported in part by the National Science Foundation grants CNS-2113805, CNS-1852134, OAC-2017564, ECCS-2010332, CNS-2037982, CNS-1563750, and CNS-1828105. We thank the reviewers for their constructive comments, which helped us improve this paper.

References

[1] Y. Zhao, and X. Chen and et al., "SmartExchange: Trading Higher-cost Memory Storage/Access for Lower-cost Computation," ISCA. 2020.
[2] S. He and H. Meng and et. al, "An efficient GPU-accelerated inference engine for binary neural network on mobile phones," Journal of Systems Architecture. 2021.
[3] N.P. Jouppi and C. Young and et al., "In-Datacenter Performance Analysis of a Tensor Processing Unit," ISCA. 2017.
[4] Y. Zhang and J. Pan and et al., "FracBNN: Accurate and FPGA-efficient binary neural networks with fractional activations," FPGA. 2021.
[5] S-E. Chang and Y. Li and et. al, "Mix and Match: A novel FPGA-centric deep neural network quantization framework," HPCA. 2021.
[6] Y. LeCun and L. Bottou and et. al, "Gradient-Based Learning Applied to Document Recognition," PROC. OF THE IEEE. London, 1998.
[7] J.T. Springenberg and A. Dosovitskiy and et. al, "Striving for Simplicity: The All Convolutional Net," ICLR (workshop track). 2015.
[8] V. Daultani and S. Chaudhury and et. al, "Convolutional Neural Network Layer Reordering for Acceleration" SASIMI, 2016.
[9] Z. Xiangyu and Z. Jianhua and et. al, "Accelerating Very Deep Convolutional Networks for Classification and Detection," CoRR. 2015.
[10] S. Karen and Z. Andrew, "Very Deep Convolutional Networks For Large-Scale Image Recognition," CoRR. 2014.
[11] C. Szegedy and W. Liu and et al., "Going deeper with convolutions," CVPR. 2015.
[12] G. Huang and Z. Liu and et. al, "Densely connected convolutional networks," CVPR. 2017.
[13] K. Alex, "CIFAR-10 and CIFAR-100 datasets," https://www.cs.toronto.edu/ kriz/cifar.html. 2010.
[14] C. Liu and B. Zoph and et al., "Progressive neural architecture search," ECCV. 2018.
[15] M.A. Ranzato and Y. Boureau and Y. LeCun, "Sparse Feature Learning for Deep Belief Networks," NIPS. 2007.
[16] K. Jarrett and K. Kavukcuoglu and et. al, "What is the best multi-stage architecture for object recognition?," ICCV. 2009.
[17] K. Alex and I. Sutskever and G.E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," Advances in Neural Information Processing Systems 25. 2012.
[18] W. Jiang and X. Zhang and et. al, "Accuracy vs. Efficiency: Achieving Both through FPGA-Implementation Aware Neural Architecture Search," DAC. 2019.
[19] D.A. Patterson and J.L. Hennessy, "Computer Organization and Design ARM Edition: The Hardware Software Interface," 2016.
[20] J.M. Rabaey and A.P. Chandrakasan and B. Nikolić, "Digital integrated circuits: a design perspective," vol. 7, 2003.
[21] "Pytorch: Tensors and dynamic neural networks in python with strong gpu acceleration," https://github.com/pytorch. 2018.
[22] N. Muralimanohar and R. Balasubramonian and N.P. Jouppi, "CACTI 6.0: A tool to model large caches," HP laboratories. 2009.
[23] S. Zhou and Y. Wu and et. al, "Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients," arXiv preprint arXiv:1606.06160. 2016.
[24] Y. Bengio, and N. Léonard and et. al, "Estimating or propagating gradients through stochastic neurons for conditional computation," arXiv preprint arXiv:1308.3432. 2013.
[25] NVIDIA, "https://www.nvidia.com/en-us/data-center/tensorcore/".
[26] C. Zhang and P. Li and er. al, "Optimizing FPGA-Based Accelerator Design for Deep Convolutional Neural Networks," FPGA. 2015.
[27] A. Manoj and C. Han and et. al, "Fused-Layer CNN Accelerators," MICRO. 2016.
[28] A. Gondimalla and N. Chesnut and et. al, SparTen: A Sparse Tensor Accelerator for Convolutional Neural Networks. MICRO, 2019.
[29] S. Han and J. Pool and et. al, "Learning Both Weights and Connections for Efficient Neural Networks," NIPS - Volume 1, 2015.
[30] Y-H. Chen and J. Emer and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," ACM SIGARCH Computer Architecture News. 2016.
[31] H. Kwon and L. Lai and et. al, "Heterogeneous Dataflow Accelerators for Multi-DNN Workloads," HPCA. 2021.
[32] S. Han and H. Mao and W. Dally, "Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding," arXiv preprint arXiv:1510.00149. 2015.
[33] H. Kartik and Y. Jiyong and et. al, "UCNN: Exploiting Computational Reuse in Deep Neural Networks via Weight Repetition," ISCA. 2018.
[34] Xilinx, "https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_2/ug888-vivado-design-flows-overview-tutorial.pdf," 2020.
[35] X. Li and G. Zhang, "Performance analysis of GPU-based convolutional neural networks," ICPP, 2016.