Expanding the Edge: Enabling Efficient Winograd CNN Inference With Deep Reuse on Edge Device

Feng Zhang, *Member, IEEE*, Ruofan Wu, Jiawei Guan, Zhen Zheng, Xiaoguang Guo, Xiao Zhang, Xiaoyong Du, and Xipeng Shen, *Member, IEEE*

***Abstract*—Deep learning on edge devices is becoming increasingly important, especially with the explosion of IoT devices. For example, the total number of devices connected to IoT reaches 29 billion in 2022. Convolutional neural networks (CNNs), as common deep learning representatives, are among the most popular neural networks in knowledge and data engineering. However, CNN employs a high degree of computing. In comparison to the training phase, the inference process is more frequently done on low-power computing equipments, such as edge devices. The limited computing resource and high computation pressure limit the effective use of CNN algorithms at the edge. Fortunately, a minimal filtering algorithm called Winograd can reduce convolution calculations by minimizing multiplication operations. We find that Winograd convolution can be accelerated further by *deep reuse* technique, which reuses the similar data and computation processes. In this paper, we propose a new inference method, called DREW, which combines deep reuse with Winograd for further accelerating CNNs. DREW handles three difficulties. First, it can detect the similarities from the complex minimal filtering patterns by clustering. Second, it reduces the online clustering cost in a reasonable range. Third, it provides an adjustable method in clustering granularity balancing the performance and accuracy. We perform evaluation on Raspberry PI and NVIDIA Jetson AGX Xavier edge devices, and experiments show that on five popular networks, 1) DREW further accelerates the Winograd convolution by an average of 8.27× speedup. Even for the highly parallel Winograd implementation, DREW still can provide 2.21× speedup. 2) When DREW is applied to end-to-end Winograd CNN inferences, DREW achieves 5.94× the average performance speedup with no (<0.4%) accuracy loss. 3) Energy consumption is an important factor for inference in practice. DREW reduces the number of convolution operations to 10% of the original operations, thus achieving up to 60% energy-efficiency benefits than the original Winograd inference.**

***Index Terms*—CNN, deep reuse, inference, winograd.**

Manuscript received 8 May 2022; revised 31 January 2023; accepted 8 April 2023. Date of publication 21 April 2023; date of current version 15 September 2023. This work was supported in part by the National Natural Science Foundation of China under Grants 62172419, 62072459, and 61732014, in part by Alibaba Group through Alibaba Innovative Research (AIR) Program, and in part by Beijing Nova Program. Recommended for acceptance by B. Glavic. *(Corresponding author: Xiao Zhang.)*

Feng Zhang, Ruofan Wu, Jiawei Guan, Xiaoguang Guo, Xiao Zhang, and Xiaoyong Du are with the Key Laboratory of Data Engineering and Knowledge Engineering (MOE), School of Information, Renmin University of China, Beijing 100872, China (e-mail: fengzhang@ruc.edu.cn; ruofanwu@ruc.edu.cn; guanjw@ruc.edu.cn; xiaoguangguo@ruc.edu.cn; zhangxiao@ruc.edu.cn; duyong@ruc.edu.cn).

Zhen Zheng is with the Alibaba Group, Hangzhou 311121, China (e-mail: james.zz@alibaba-inc.com).

Xipeng Shen is with the Computer Science, North Carolina State University, Raleigh, NC 27695 USA (e-mail: xshen5@ruc.edu.cn).

Digital Object Identifier 10.1109/TKDE.2023.3269017

I. Introduction

DEEP learning has shown successes and gained popularity in data science applications [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], especially with the booming of IoT devices. For example, by 2022 the total number of devices connected to IoT worldwide reaches 29 billion and will increase to about 75 billion by 2025 [11]. Convolutional neural networks (CNNs), as representatives of deep learning networks, draw much attention in knowledge and data engineering. Different from the training process, inferences of CNN are widely applied at the edge in industry and face a high demand for performance optimization [12], [13], [14], [15], as shown in Fig. 1. Note that accelerators with high computing power are usually too expensive for companies to apply to inference workloads for mass deployment. In industry, the CNNs are usually trained on HPC clusters, while the inference could be conducted on less powerful devices, such as mobile processors at the edge [16], [17]. Energy efficiency is another concern, since edge computing usually happens in resource-constrained environments. Due to high compute density, it is important to optimize the inference process at the edge, especially for industry usage. The key to improving the inference performance of CNN models is to accelerate the convolutional layers, which are computation-intensive and dominate the total execution time. In this paper, we study CNN accelerations on edge devices.

Among massive CNN optimization techniques, Winograd convolution [18] has been proved to be an effective method. Employing Winograd minimal filtering algorithm reduces the arithmetic complexity of convolution operations by at least 2.25× [18] theoretically, saving substantial time and energy. The majority of modern deep learning libraries, including Nvidia cuDNN [19] and Intel oneDNN (previously known as MKL-DNN) [20], enable Winograd convolution for CNNs. Additionally, several attempts have been made to accelerate Winograd convolution via increased hardware efficiency [21], [22]. Hence, Winograd is a potential optimization method to accelerate CNN computation on edge devices.

There is a large literature of Winograd convolution. However, prior research on Winograd convolution concentrated on how to improve the algorithm's performance on certain hardware platforms, such as GPUs [19], [22], rather than the algorithm's structure. We find that, rather than performing typical code optimizations, a unique approach called *deep reuse* [10] can uncover and leverage reused calculations to accelerate convolutions.

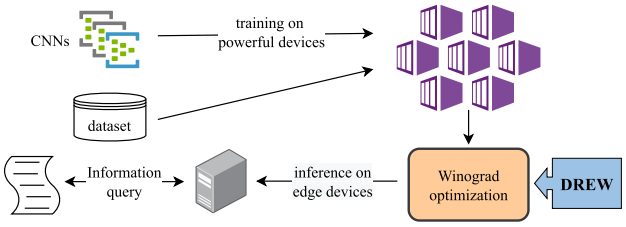


Fig. 1. CNN in industrial edge devices.

This approach reuses intermediate results in CNN inference by recognizing similarities among neuron vectors, saving both space and time on the fly [23]. The present deep reuse approach, unfortunately, is limited to GEMM-based convolution. The performance of CNN inference could be considerably enhanced if deep reuse is applied to the Winograd convolution. This is especially beneficial for edge situations.

Integrating deep reuse with Winograd convolution at the edge faces three major challenges. First, Winograd convolution involves fixed minimal filtering patterns, so there is no direct neuron vector to extract in the Winograd algorithm. Second, deep reuse is an online process in which the similarity detection process among neuron vectors happens during the inference time. Accordingly, it poses a tighter constraint on the introduced overhead compared to the conventional convolution. Third, the limitations of minimal filtering patterns also prevent us from adjusting the reuse granularity as flexibly as the original deep reuse does. Because deep reuse is a lossy optimization, to provide the choice to trade-off between performance and accuracy loss, we need to propose a novel design to adjust reuse granularity at the edge.

We develop a new inference method, called *DREW*, which can effectively combine **d**eep **re**use with **W**inograd convolution on edge devices, as shown in Fig. 1. DREW solves the challenges above and brings huge performance improvements. First, we design a novel approach to leverage the neuron similarities in Winograd convolution, which has been proved to have great potentials. Second, to minimize the runtime overhead introduced by deep reuse, we develop a novel clustering process, which causes only a small proportion of time compared to the overall operation time and reduces the space overhead on the fly. Third, we extend our approach to make it adjustable in clustering granularity, and leave the adjustment between performance and accuracy to users to meet their different needs. Moreover, we make our solution, DREW, a library for users to easily apply our work. Our preliminary work has been presented in DREW [24], which provides only a simple design without considering the edge situation. In this work, we perform DREW at the edge.

We evaluate DREW on Raspberry PI and NVIDIA Jetson AGX Xavier edge devices with five popular neural networks: LeNet-5 [25], CifarNet [26], VGG-11 [2], VGG-16 [2], and SqueezeNet [27]. For single-layer performance, DREW achieves 8.27× speedup on average compared to the Winograd convolution without deep reuse. Even for the highly parallel Winograd implementation, DREW can still provide 1.13× to 4.36× performance improvement. For end-to-end performance,

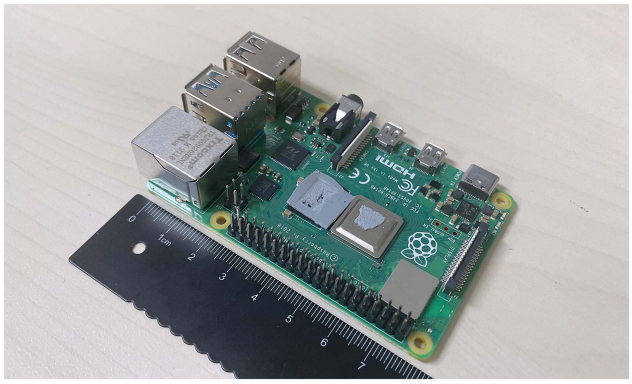


Fig. 2. Raspberry PI 4 Model B.

DREW achieves 5.94× performance improvement with little (<0.4%) accuracy loss, and for parallel implementation, DREW still maintains 1.16× to 2.75× performance improvement. With detailed analysis, the convolution operations can be reduced to an average of 10% of the original computations and take up 33% to 55% of the original execution time. Energy efficiency is another important factor for devices executing CNN inference. We also evaluate DREW from the energy perspective. As the computation amount is greatly reduced, DREW improves the average energy efficiency of CNN inference by up to 62%.

In summary, this work makes the following contributions.

- It points out that deep reuse can be efficiently combined with Winograd convolution at the edge for the first time. This work proposes a new inference optimization method, called DREW, which can detect and exploit input similarities among Winograd minimal filtering computation patterns.
- It designs a novel clustering process for DREW at the edge, which reduces online cost in inference. It extends DREW to adjust the clustering granularity, allowing users to balance the trade-off between accuracy and efficiency.
- It validates the efficacy of DREW and demonstrates its significant performance and energy benefits with almost no accuracy loss on the edge device.

II. Background

In this section, we first introduce edge computing, followed by the Winograd convolution, and then show the deep reuse technology.

A. Edge Computing

Edge computing brings data processing closer to the source, and edge devices consume data locally. With its advantages of reducing data transmission, improving service quality, protecting user privacy, and relieving cloud computing pressure, edge computing has become an important solution to break through the bottleneck of emerging Big Data problems. For example, the edge device Raspberry PI 4 Model B [28], as shown in Fig. 2, is a 86×56 mm small device based on a high-performance 64-bit quad-core ARM Cortex-A72 64-bit SoC, 1/2/4 GB of

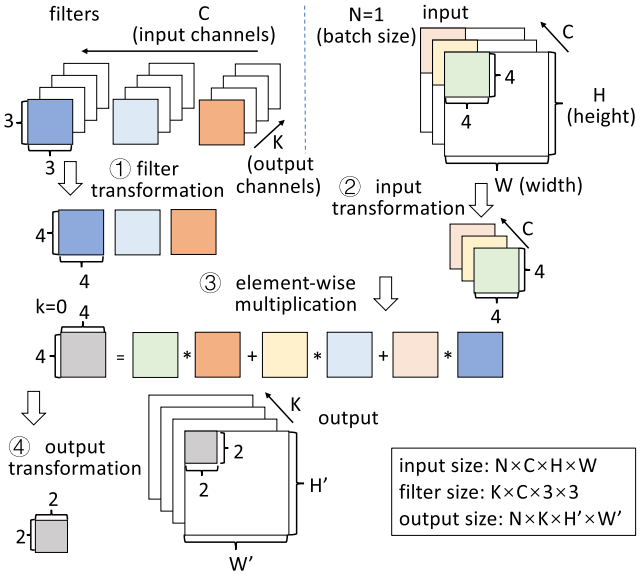


Fig. 3. The workflow of Winograd convolution.

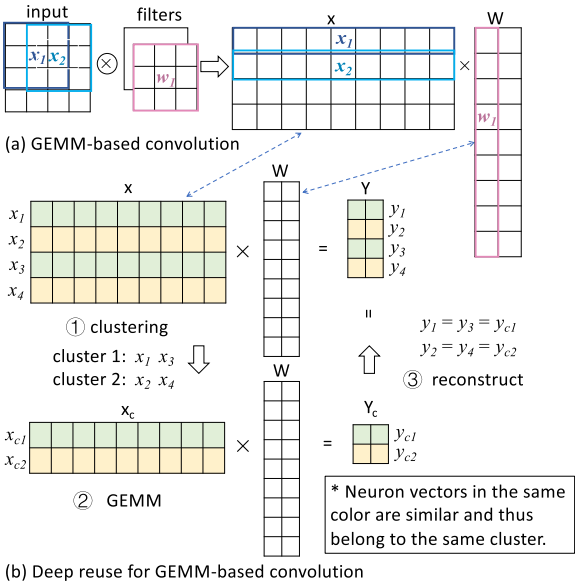


Fig. 4. The illustration of GEMM-based convolution and the workflow of deep reuse.

RAM, one Gigabit Ethernet interface, dual-band 2.4/5.0 GHz wireless LAN, Bluetooth 5.0, two USB 2.0 ports, two USB 3.0 ports, two micro-HDMI connectors, and one microSD card slot. Such a functional edge device costs less than $55, with power consumption less than 6.4 W.

Edge devices, such as Raspberry Pi, are an apt choice for IoT Applications [29]. Countless Internet-based projects are already using them to create tablets [30], laptops [31], robots [32], and smart mirrors [33]. Although they can be applied to common applications, for popular convolutional neural networks, these devices still lack capacity. For example, Raspberry PI 4 Model B is powered by 1.5 GHz quad-core processor with low computing power, which requires additional optimization to execute convolutional neural networks, especially the inference process.

B. Winograd Convolution

The Winograd convolution is a kind of convolution algorithm that employs the Winograd minimal filtering algorithm, resulting in fewer arithmetic operations compared to the original implementation [34]. The Winograd minimal filtering algorithm, denoted as $F(m\times m, r\times r)$, computes $m\times m$ outputs with a $r\times r$ filter, and reduces the number of multiplications from $(m\times r)^2$ to $(m+r-1)^2$. In this paper, we use a common case of $F(2\times 2, 3\times 3)$ for application, which has also been used in previous studies [20], [21], [22].

Workflow. The workflow of the convolutional layer with $F(2\times 2, 3\times 3)$ Winograd minimal filtering algorithm is shown in Fig. 3. First, each 3×3 filter is performed by a *filter transformation* (Step 1) to a 4×4 transformed filter. Second, the input images or feature maps are divided into *tiles* of size 4×4, with 2 elements overlapping between neighboring tiles. Each tile is performed by an *input transformation* (Step 2) to a 4×4 transformed input tile. Third, it executes *element-wise multiplication* (Step 3) with the filter of the corresponding input channel and accumulation along input channels. Fourth, it performs *output transformation* (Step 4) for each pre-transformed output tile (the result of element-wise multiplication and accumulation) to a 2×2 output tile.

Difference From Direct Convolution. Winograd convolution is significantly different from direct convolution. First, Winograd convolution needs to load a 4×4 tile before performing computation, instead of directly loading an element for a multiply-add operation. Second, Winograd convolution needs to perform an input transformation and an output transformation before and after element-wise multiplication. Third, Winograd convolution takes 2×2 tile as the smallest unit to obtain computational results, while direct convolution can obtain the computational result of one output element directly.

Winograd CNN Inference. For online Winograd CNN inference, the filter transformation can be finished at preprocessing time once. Therefore, the inference time mainly comes from input transformation, element-wise multiplication, and output transformation, which are the emphasis of our optimization. Among these operations, there can be a large number of similar neurons, where data reuse optimization can be applied. This is beneficial for inference at the edge.

C. Deep Reuse

Deep reuse is a kind of data science optimization for accelerating CNN inferences by detecting and utilizing runtime similarities among input data [10], [23]. To compute the convolutional layer, the common practice is to unfold the input images and filters into input matrix x and weight matrix W, and then perform General Matrix Multiplication (GEMM) with two matrices, as shown in Fig. 4(a). The idea of deep reuse is that strong similarities exist among *neuron vectors*. Here, a neuron vector is composed of several consecutive elements in

a row of the unfolded input matrix x. Therefore, the neuron vectors can be clustered into a small number of groups, and the computation results for the cluster centroids can be reused by all neuron vectors in clusters.

Workflow. We show the deep reuse workflow in Fig. 4(b). First, the neuron vectors, which are the rows of the input matrix x, are clustered and represented by the cluster centroids. In the instance of Fig. 4(b), the neuron vectors are clustered into two centroids. Second, the input-weight multiplication is transformed into centroid-weight multiplication. Finally, the results are reconstructed using the computation results of cluster centroids.

Clustering Method. Deep reuse uses Locality Sensitive Hashing (LSH) [35] as the clustering method to detect similarities among neuron vectors because LSH can deliver good clustering results and does not introduce excessive overhead to inferences. LSH ensures that the computation savings serve its purpose for performance improvement without accuracy loss. For each input vector *x*, a *hash function h* is determined by a random vector *v* in the following (1):

$$h_v(x) = \begin{cases} 1 & if \quad v \cdot x > 0 \\ 0 & if \quad v \cdot x \leq 0 \end{cases} \tag{1}$$

With H random vectors, LSH maps an input vector into a bit vector with 2^H possibilities. The input vectors that are close to each other have a high probability to be hashed into the same bit vector. Thus, the roughness of the clustering can be adjusted by the number of hash functions. After LSH being applied to deep reuse, the integer value of the bit vector can be used as a cluster ID. Then, the cluster centroids are computed using the neuron vectors with the same cluster ID for retrieving them for later computation. Ning et al. [23] define the *remaining ratior*$_c$ to measure reusable potential, which is the ratio of the number of clusters attained after LSH to the total number of neuron vectors. A smaller remaining ratio indicates larger computation savings.

Difficulties in Combining Winograd and Deep Reuse. If we can combine deep reuse with Winograd convolution effectively, the CNN inference can be further accelerated significantly. However, deep reuse cannot be directly applied to Winograd convolution. Applying deep reuse to Winograd convolutions involves complexities. For example, the tiles in Winograd-based convolution are fixed, which is to say, we cannot divide the tiles into different vectors for clustering to retain the advantages of the original Winograd algorithm, because results are obtained from tiles.

III. Revisiting Winograd Convolution

In this section, we first analyze the reuse opportunities on Winograd convolution. Second, we show our observations and insights. Third, we discuss the importance and benefits of our work.

A. Opportunities

We explore the reuse opportunities of combining deep reuse and Winograd-based convolutions and show the Winograd-based convolution in Fig. 5. We have the following insights.

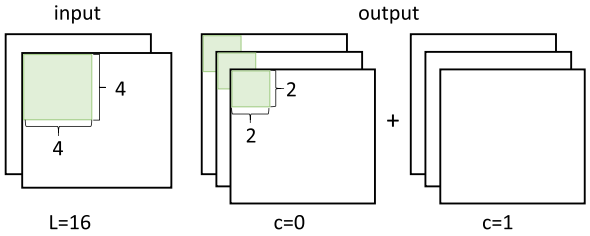


Fig. 5. Tile in Winograd-based convolution. The input tile size 4x4 and output tile size 2 × 2 are defined by F(2 × 2,3 × 3).

Insight 1: The Neuron Similarities Exist in Winograd Convolution. Based on our observation, neuron similarities exist in Winograd convolution. For Winograd-based convolution, the 4×4 tiles are local neurons as shown in Fig. 5. Due to the continuity in images or feature maps that CNN often targets, it has been proved that neighbor neurons have extremely strong similarities [36]. Hence, there is a high probability that similarities exist among such small tiles in Winograd convolution. This provides us with a great opportunity to further accelerate CNN inference, and thus we can use the similarity between tiles in Winograd to reduce the amount of computation to save time.

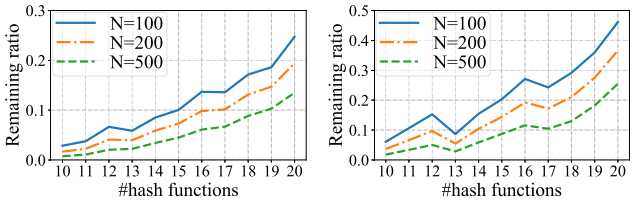
Insight 2: Deep Reuse is Worth to be Applied to Winograd Convolution, Because it has Much Higher Performance Potential Than That in Conventional Convolution. Winograd convolution reduces the multiplication computation, and the tile in Winograd convolution is suitable for data reuse. Assume that there are large similarities among tiles of batched input, by combining deep reuse and Winograd with proper granularity, we can further significantly reduce the amount of computation for the CNN inference and bring performance improvements.

B. Observation

> *Finding:* Experimental observations prove our assumption on tile similarities: there is huge potential of performance improvement for combining deep reuse with Winograd convolution.

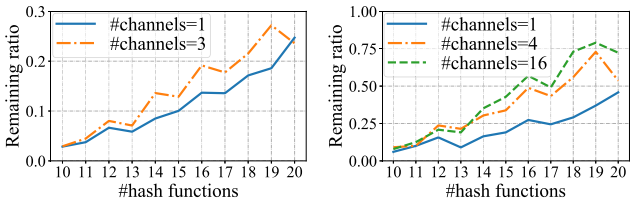
To prove our assumption on tile similarities, we conduct a series of experimental analysis and draw the conclusion that there is huge potential for performance improvement of combining deep reuse and Winograd convolution. We use the trained model of CifarNet [26] on CIFAR10 [37], and run its inference for illustration. We apply LSH with different numbers of hash functions to the two convolutional layers of *Conv1* and *Conv2* and record the remaining ratio after clustering. Note that a larger number of hash functions means a more fine-grained clustering, and the remaining ratio indicates the reusable potentials, as discussed in Section II-C. Figs. 6 and 7 show the experimental results of *Conv1* and *Conv2* in CifarNet.

Observation 1: Similarities Exist in Single Channel. We detect the similarities among tiles in each channel with different batch sizes and report the average remaining ratio of each channel. The results are demonstrated in Fig. 6, which shows that the two



(a) Conv1 (b) Conv2

Fig. 6. The remaining ratio of the different numbers of hash functions with different batch sizes in a single channel. N represents the batch size.



(a) Conv1 (b) Conv2

Fig. 7. The remaining ratio with different numbers of hash functions and different numbers of channels.

convolutional layers exhibit similar trends. The remaining ratio (introduced in Section II-C) reaches an average of 0.084 in the first convolutional layer and 0.139 in the second convolutional layer. They both increase along with the increase of hash size and the decrease of batch size.

Observation 2: Similarities Exist Between Different Channels. We detect the similarities among tiles of multiple channels with a fixed batch size of 100 (we treat the tiles of several consecutive channels as the neuron vectors) and present the results in Fig. 7. Assume that the input to the convolutional layer has C channels. We evenly divide the C channels into x parts so each part has C/x channels. We treat the tiles of consecutive C/x channels as the neuron vector and apply LSH to them in each part. The results show that tiles of multiple channels still generate a small remaining ratio, especially when the number of hash functions is small. Generally, fewer channels lead to a smaller remaining ratio.

To sum up, large data similarities exist among tiles of Winograd convolution, which provides reuse opportunities.

C. Challenges

Winograd has been proved to be an important method accelerating convolution inference, and we find that the tile in Winograd convolution is suitable as the object we reuse, as discussed above. Since the observations have proved that there is large similarity among tiles of batched input, a large amount of computation can be saved by leveraging the similarity. Thus, higher performance speedups and energy savings of Winograd convolution can be achieved, which are of great benefit to the inference devices.

However, combining deep reuse and Winograd convolution requires addressing the following three challenges.

Challenge 1: Algorithm Design. The computation process in Winograd convolution is complicated: Winograd convolution involves fixed minimal filtering patterns, whose neuron vectors *cannot* be extracted directly. Consequently, an appropriate method needs to be designed for reuse, exploiting the similarities and saving computations.

Challenge 2: Clustering Overhead. Deep reuse is an online process in which the similarity detection process among feature maps happens during the inference time. Accordingly, it poses a tight constraint on clustering time: the time overhead introduced by LSH, including detecting similarities, computing cluster centroids, and retrieving for reuse, has to be restricted to a minimal range. In particular, logics in retrieving the clustering result for reuse need to be parallelized for efficiency, making this challenge even more difficult to solve.

Challenge 3: Cost-Benefit Trade-off. The size of tiles in Winograd convolution is fixed, so we cannot flexibly adjust the clustering granularity by changing the length of neuron vectors. To further optimize the performance of Winograd convolution and balance the trade-off between performance and accuracy loss, a novel method to adjust clustering granularity needs to be designed.

IV. DREW Overview

We show in Section III that strong similarities exist among input tiles of each channel in Winograd convolution, which provides us the opportunity to save computations by reusing the computed results of a small number of tiles. In this section, we first elaborate on our idea of applying deep reuse in Winograd convolution. Then, we show an example and present our solutions to the challenges listed in Section III-C.

Idea. Revisiting the process of Winograd convolution mentioned in Section II-B, we can see that the workflow of Winograd convolution includes 1) filter transformation, 2) input transformation, 3) element-wise multiplication, and 4) output transformation. We group the input tiles of each channel into clusters and compute the cluster centroids. Then, we perform input transformation, K element-wise multiplication, and K output transformation on these centroid tiles. Finally, we accumulate the corresponding centroid tiles of each input channel to produce output. Note that the accumulation along input channels happens on element-wise multiplication in the original Winograd convolution. We leave it to the last for saving addition operations.

Example. We show an example in Fig. 8 for processing a $1\times2\times6\times6$ input with $2\times2\times3\times3$ filters. The filters have been transformed during the preprocessing time. First, after clustering, four input tiles are grouped into two clusters of each channel respectively. The four input tiles can be represented by the two cluster centroids. Second, four centroid tiles of all channels are transformed in a way like the input transformation in the Winograd algorithm. Third, each transformed centroid tile is element-wise multiplied by the filters of two output channels. Fourth, the results of multiplication are transformed in a way like the output transformation in the Winograd algorithm. Fifth, we accumulate the output along input channels, and obtain all tiles in the final output from the computed centroid tiles.

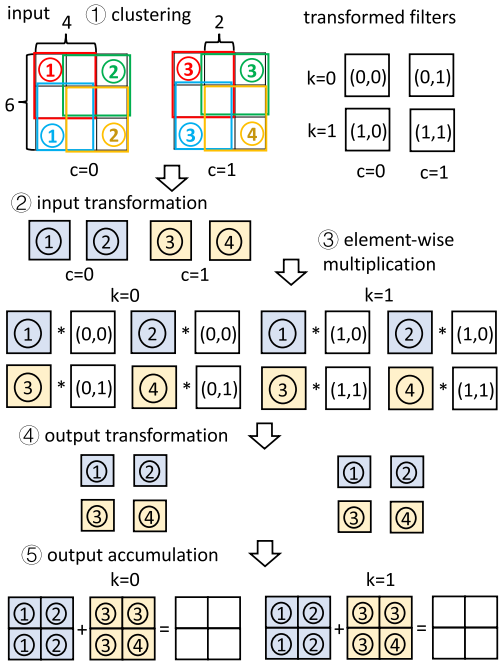


Fig. 8. Example of applying deep reuse to Winograd.

Solutions to Challenges. We develop DREW to solve the challenges listed in Section III-C. First, we design a method to leverage the neuron similarities in Winograd convolution in which the tile size 16 is the smallest clustering granularity (Sections IV and V-A). Second, to minimize the time overhead introduced by clustering, we choose the fast LSH as the clustering method. However, we design a new method to retrieve cluster centroids, which is more suitable for modern processors and can minimize the space overhead (Section V-B). Third, we extend the algorithm to reuse tiles of the input channels, which allows users to tune the clustering granularity for the trade-off between accuracy and time savings (Section V-C).

Novelty. Based on the solutions above, our work makes the following novel contributions. First, we develop new algorithms to detect similarities in the complex filtering patterns of Winograd (Section V-A). Second, we provide novel clustering designs in combing deep reuse and Winograd to reduce online costs within a reasonable range (Section V-B). Third, we provide a novel method for users to adjust the clustering granularity, which can balance the trade-off between performance and accuracy (Section V-C). Then, we develop a fine-tuning process to maintain high accuracy (Section VII-D).

Discussion. After deep reuse is applied to Winograd convolution, the computations of input transformation, element-wise multiplication, and output transformation in Winograd convolution can be greatly reduced. Note that we cannot eliminate the input and output transformations between layers. The input and output transformations are integral parts of the convolutional computation formulated as the Winograd minimal filtering algorithm, and eliminating them would ruin the results of convolutional computation.

TABLE I
SUMMARY OF NOTATIONS

Symbol	Meaning
$N \times C \times H \times W$	Input size
$K \times C \times 3 \times 3$	Filter size
$N \times K \times H' \times W'$	Output size
$P = N\lceil H'/2 \rceil \lceil W'/2 \rceil$	Number of tiles
$I_{n,c,i,j}$	Input tile
i	Tile index in height
j	Tile index in width
$\hat{I}_{n,c,i,j}$	Transformed input tile
$\hat{F}_{k,c}$	Transformed filter tile
$\hat{O}_{n,k,i,j}$	Pre-transformed output tile
$O_{n,k,i,j}$	Output tile
$\hat{H}$	Number of hash functions
$T \in R^{\hat{H} \times 16}$	Hash table with $\hat{H}$ random vectors
$V_{n,c,i,j}$	Neuron vector flattened from input tile
$P_{n,c,i,j}$	Bit vector after LSH for input tile
$p_{n,c,i,j}$	Integer value of bit vector $P_{n,c,i,j}$
$b_{n,c,i,j}$	Bucket ID of neuron vector after LSH
C_b	Centroid vector of bucket b
N_b	Number of neuron vectors of bucket b
$\hat{C}_b$	Transformed centroid vector of bucket b
$\hat{D}_{b,k}$	Element-wise multiplied centroid vector
$D_{b,k}$	Transformed output
r_c	Remaining ratio
L_{cb}	The number of tile channels for clustering

V. DETAILED DESIGN

After presenting the idea of combining deep reuse and Winograd convolution in Section IV, we introduce the detailed design of DREW in this section. First, we present the basic workflow of DREW. Second, we delve into the design of clustering and extend the clustering granularity to multiple channels. Third, we show the parallel design of DREW.

A. Deep-Reuse Winograd

Preliminaries. Before illustrating the design of DREW, we first review the notations and the original Winograd convolutions. We list the notations used in our work in Table I.

Winograd Presentation. The original Winograd convolution can be written in the following formulations. For $N \times C \times H \times W$ inputs with $K \times C \times 3 \times 3$ weight filters in Winograd convolution (Section II-B), filter transformation $\hat{F} = GFG^T$ is finished in the preprocessing time, where G is a 4×3 matrix defined in Winograd minimal filtering algorithm. During the inference time, the input images or feature maps are divided into $P = N\lceil H'/2 \rceil \lceil W'/2 \rceil$ tiles, where H' and W' are the height and width of outputs. An input tile $I_{n,c,i,j}$ in channel c is transformed by $\hat{I}_{n,c,i,j} = B^T I_{n,c,i,j} B$, where B is a 4×4 matrix defined in Winograd minimal filtering algorithm. Then, the transformed tile $\hat{I}_{n,c,i,j}$ is element-wise multiplied with K transformed weights and accumulated with the results of the tiles in the same position in other channels, which is $\hat{O}_{n,k,i,j} = \sum_{c=0}^{c-1} \hat{F}_{k,c} \odot \hat{I}_{n,c,i,j}$. Finally, the output of $O_{n,k,i,j}$ in channel k can be obtained from the output transformation $A^T \hat{O}_{n,k,i,j} A$, where A is a 4×2 matrix defined in Winograd minimal filtering algorithm.

Combining Deep Reuse and Winograd Convolution. In DREW, the 2D batched *deep reuse for Winograd convolution*

can be written in the following five steps. Note that the filters have been transformed during the preprocessing time.

Step 1: Clustering. For each 4×4 input tile $I_{n,c,i,j}$, $V_{n,c,i,j}$ is the 1D neuron vector flattened from $I_{n,c,i,j}$. Each neuron vector $V_{n,c,i,j}$ is projected as a bit vector $P_{n,c,i,j}$ by $\hat{H}$ hash functions, as shown in (2), where $p_{n,c,i,j}$ is the integer value of the bit vector $P_{n,c,i,j}$. Note that $T \in R^{\hat{H}\times 16}$ is the hash table with $\hat{H}$ random vectors.

$$P_{n,c,i,j} = TV_{n,c,i,j} \in R^{\hat{H}} \tag{2}$$

Then, for each input channel c, the identical integer values of neuron vectors are mapped to the same bucket to obtain the bucket ID $b_{n,c,i,j}$ and the bucket size N_b. Note that the bucket ID b of neuron vectors in different input channels is different because we cluster the tiles of each channel respectively. Finally, the cluster centroid of each bucket b is calculated in (3).

$$C_b = \sum_{V_{n,c,i,j} \in bucket_b} V_{n,c,i,j}/N_b \tag{3}$$

Step 2: Input transformation. For each centroid vector C_b, we perform input transformation, as shown in (4).

$$\hat{C}_b = B^T C_b B \tag{4}$$

Step 3: Element-wise multiplication. For each centroid vector C_b of input channel c and each output channel k, we perform element-wise multiplication, as shown in (5).

$$\hat{D}_{b,k} = \hat{C}_b \odot \hat{F}_{c,k} \tag{5}$$

Step 4: Output transformation. For each centroid vector C_b, we perform output transformation, as shown in (6).

$$D_{b,k} = A^T \hat{D}_{b,k} A \tag{6}$$

Step 5: Output accumulation. For each output tile and each output channel k, we perform output accumulation, as shown in (7).

$$O_{n,k,i,j} = \sum_{c=0}^{C-1} D_{b_{n,c,i,j},k} \tag{7}$$

B. Clustering Design

We analyze our clustering design of *Step 1* mentioned in Section V-A. Since Winograd is an online process, Winograd poses a tight constraint on clustering time. The clustering process includes LSH projection, the computation of the integer value of bit vectors, bucket mapping, and centroid calculation. To minimize the time overhead caused by deep reuse, we mainly optimize the following two substeps, LSH projection and bucket mapping.

LSH Projection. The LSH has been introduced in Section II-C. However, the LSH cannot be used directly. For example, we need to adjust the number of hash functions in Winograd applications. The design of LSH projection is as follows.

1) Detailed design. For our batched *deep-reuse Winograd* in DREW, instead of indexing the vectors one by one, we perform LSH projection by an $H\times 16\times CP$ GEMM at one time. Consequently, we convert it into a GEMM process. Then, we compute the integer value of each bit vector, which is the projected neuron vector, for bucket mapping. After LSH is applied, the input vectors with small distances have a higher probability of being hashed to the same bit vector, so these input vectors are easier to be mapped to the same bucket in our application.

2) Configuration. To obtain the best configuration for clustering using LSH, we need to adjust LSH to find a solution that can recover accuracy while minimizing the amount of computation. The number of hash functions $\hat{H}$ is a parameter for clustering configuration. It has been proved that $\hat{H}$ mainly affects r_c, where r_c is the remaining ratio $\frac{|C|}{N}$ [23]. If we use a large $\hat{H}$ for LSH, then we have a high probability of getting more buckets through clustering. Accordingly, fewer vectors will be assigned into each bucket, which will result in an increase in r_c and less computation reduction. Therefore, we prefer a small number of hash functions. However, if $\hat{H}$ is too small, the variance in a bucket will be large, which will bring a loss to the accuracy of CNN. Consequently, we need to make a trade-off between the amount of calculation and accuracy, and then use the optimal parameter configuration to achieve good results. More detailed analysis can be found in Section VII-E.

3) Complexity analysis. With $\hat{H}$ hash functions, the total computational complexity of clustering is $O(C{\cdot}P{\cdot}\hat{H})$, which can also be presented as $O(C{\cdot}N{\cdot}H{\cdot}W{\cdot}\hat{H})$.

Bucket Mapping. Bucket mapping uses the integer value of each bit vector to map similar vectors to the same bucket (cluster) and thus clustering results are obtained. This is the most difficult part to be parallelized in combining deep reuse and Winograd in DREW, because we need to record the buckets with their related vectors and calculate the bucket size.

1) Analysis. Before showing our bucket mapping design, we first revisit the bucket mapping in [23], which treats the integer value of the bit vector as cluster ID (to distinguish it from ours, we denote ours as bucket ID). Utilizing these cluster IDs minimizes the time overhead in a conflict-free hash. However, such a solution *cannot* be used in our situation. The space for $2^{\hat{H}}$ cluster centroids needs to be allocated before centroid calculation; if we adopt this method, we need to allocate $C\times 2^{\hat{H}}$ neuron vectors for later element-wise multiplication. This would be a huge space overhead and would become a burden especially on the industrial hardware where CNN inferences commonly run.

2) Algorithm design. We develop a novel bucket mapping strategy in DREW, as shown in Algorithm 1. First, DREW needs to map the tiles $I_{n,c,i,j}$ to the buckets $b_{n,c,i,j}$. DREW iterates over $\lceil H' \rceil \times \lceil W' \rceil$ values in C channels, as shown from Lines 6 to 8. Then, for batch n, DREW traverses each 4×4 input tile $I_{n,c,i,j}$ in C channels, and obtains the tile's integer value from the bit vector *proj_vectors* in Line 9.

Second, DREW increases the newly discovered bucket ID in order and counts bucket size. With channel c and hash $value$, we can obtain the bucket ID b from $hash\_map$ in Line 10.

If the bucket is first met, which means that b is equal to 0 as shown in Line 11, then, no vector whose hash value is equal to

$value$ has yet appeared in channel c. Hence, a new bucket needs to be created. DREW records the bucket ID corresponding to the hash value in Line 12, and sets the number of vectors in the bucket to 1 in Line 13. Then, DREW records the vector ID for subsequent solving of the cluster center in Line 14 and increases the bucket number in Line 15.

If the bucket for the current hash value already exists, which means that b is greater than 0 in Line 16, we need to increase the bucket count by 1 and record the vector ID, as shown from Lines 17 to 19. When all channels have been processed, we save the number of total buckets in Line 20.

Third, DREW computes the cluster centroid of each bucket from Lines 22 to 27. In detail, DREW traverses the vectors in each bucket, and sets the centroids with the average of *input_vectors*.

3) Complexity. We analyze the complexity of Algorithm 1. With such design, we only need $O(\sum N_b)$ space for clustering and later computation. Because the integer values of vectors provide a hash-map with an $O(1)$ lookup complexity in nature, we can finish bucket mapping in $O(C{\cdot}P)$ complexity, whose time proportion in the whole clustering phase is small (compared with LSH projection and centroid calculation). Experiments also show that the time of this part is acceptable (Section VII-F).

C. Clustering Granularity

To further optimize the performance of DREW, we extend the clustering granularity to tiles of multiple channels. This is regarded as a user-defined parameter to trade-off between time savings and accuracy loss.

Limitations in the Algorithm. First, because it is uncertain which bucket each neuron vector is mapped to, the memory access to $D_{b_{n,c,i,j}}$ in output accumulation is discontinuous. The output accumulation phase in our basic algorithm needs a long time in the whole process. This situation could be alleviated if we find a way to reduce such memory access pressure. Second, if we only define the tile of one channel as the clustering granularity, users cannot adjust the performance and accuracy based on their requirements.

Clustering Granularity Design. We solve such limitations by clustering tiles of multiple channels in DREW. We find that tiles of multiple channels also have similarities with each other and these tiles even reach a smaller remaining ratio when the number of hash functions is small. If we cluster tiles of multiple channels and reuse the computation results, the results of element-wise multiplication in these channels can be accumulated in the element-wise multiplication phase and the amount of output accumulation is reduced. Therefore, the performance becomes higher due to the smaller remaining ratio and fewer discontinuous memory accesses.

D. Parallelism

In this section, we first discuss the parallelism and complexity of DREW. To facilitate the analysis of computation savings, we uniformly denote the height and width of the input and output as H and W. Then, we analyze the properties of clustering parameters according to the complexity.

Algorithm 1: Bucket Mapping.

```
1  Function Bucket_Mapping(input_vectors,
     proj_vectors, C, H', W', n, Ĥ):
2      MEMSET(*hash_map, 0)
3      MEMSET(*bucket_count, 0)
4      total_bucket = 0
5      cur_bucket = 0
6      for c in [0, C] do
7          for i in [0, ⌈H'⌉] do
8              for j in [0, ⌈W'⌉] do
9                  p = getHashValue(proj_vectors, n, c, i, j)
10                 b = hash_map[c×(1 << Ĥ) + p]
11                 if b == 0 then
12                     hash_map[c×(1 << Ĥ) + p] = cur_bucket
13                     bucket_count[cur_bucket] = 1
14                     bucket2vecid[cur_bucket].pushback(i∗
                         ⌈H'⌉ + j + c ∗ ⌈H'⌉ ∗ ⌈W'⌉)
15                     cur_bucket+ = 1
16                 end
17                 else
18                     bucket = b − 1
19                     bucket_count[bucket]+ = 1
20                     bucket2vector[bucket].pushback(i ∗
                         ⌈H'⌉ + j + c ∗ ⌈H'⌉ ∗ ⌈W'⌉)
21                 end
22             end
23         end
24     end
25     total_bucket = cur_bucket
26
       // compute the cluster centroids
27     for b in [0, total_bucket] do
28         vec_num = bucket_count[b]
29         for i in [0, vec_num] do
30             vecid = bucket2vecid[b][i]
31             for d in [0, 16] do
32                 centroid[b ∗ 16 + d]+ =
                     (input_vectors[vecid∗16+d]/vec_num)
33             end
34         end
35     end
36 return centroid
```

Parallel Design. All the five steps of DREW described in Section V-A can be parallelized. For the first step of clustering, the LSH projection can be treated as a GEMM and is thus processed in parallel. The integer value of each projected bit vector and the computations for each bucket centroid can also be done in parallel. For the second step of input transformation, the third step of element-wise multiplication, and the fourth step of output transformation, the computations for each centroid vector can be executed in parallel and we process these three steps within a loop to avoid unnecessary memory accesses. For the fifth step of output accumulation, we retrieve the results of centroid vectors of C channels and accumulate them for each output tile in parallel.

Complexity. The computational complexity of the original Winograd convolution is $O(N{\cdot}H{\cdot}W{\cdot}C{\cdot}K)$ [18]. Assume that input channels are divided into N_{cb} channel blocks and each channel block contains tiles of L_{cb} channels (C=$N_{cb} \cdot L_{cb}$). From the previous analysis of LSH in Section V-B, the total

computational complexity of clustering is $O(C{\cdot}N{\cdot}H{\cdot}W{\cdot}\hat{H})$. If the neuron vectors can be grouped into $|\hat{C}|$ clusters, the average number of clusters $|\hat{C}|_{cb,avg}$ is $\frac{1}{N_{cb}}\sum_{j=1}^{N_{cb}}|\hat{C}|_{cb,j}$. The remaining ratio of Winograd with deep reuse r_c is $\frac{|\hat{C}|_{cb,avg}}{CP}$. Then, the computational complexity of the Winograd phase except for output accumulation is $O(r_c{\cdot}N{\cdot}H{\cdot}W{\cdot}C{\cdot}K)$ and the computational complexity of output accumulation is $O(N_{cb}\cdot N\cdot H\cdot W\cdot K)$. Therefore, the overall computational complexity of DREW is $O\left(\left(\frac{\hat{H}}{K}+r_c+\frac{1}{L_{cb}}\right)\cdot N\cdot H\cdot W\cdot C\cdot K\right)$.

Properties. There are two clustering parameters to adjust in DREW: clustering granularity (the number of tile channels L_{cb}) and the number of hash functions ($\hat{H}$). They affect the time savings by reuse and accuracy loss. With the computational complexity analysis with different parameter combinations, we observe the following properties when combining deep reuse and Winograd convolution:

- When $\hat{H}$ remains unchanged, a smaller granularity (smaller L_{cb}) generally leads to a smaller reuse-caused accuracy loss. Meanwhile, a smaller L_{cb} leads to more additional operations in output accumulation.
- When L_{cb} remains unchanged, more hash functions (larger $\hat{H}$) generally incur smaller reuse-caused accuracy loss. Meanwhile, a larger $\hat{H}$ causes a larger number of clusters and thus a larger r_c.
- When L_{cb} is large, $\hat{H}$ affects the reuse-caused accuracy loss and r_c more than L_{cb} does. When L_{cb} is small, the change of L_{cb} affects the reuse-caused accuracy loss and r_c more than $\hat{H}$ does.
- An appropriate combination of L_{cb} and $\hat{H}$ can reduce $\left(\frac{\hat{H}}{K}+r_c+\frac{1}{L_{cb}}\right)$, which is a coefficient in DREW's complexity, thus resulting in more computation savings.

VI. Implementation

We build a library called DREW for three purposes. First, DREW can show users the benefits of combining deep reuse technology and Winograd convolution, which could shed light on the research of applying data science technologies to machine learning. Second, to increase the compatibility of DREW with other deep learning applications, we implement a complete CNN pipeline in DREW, including a series of typical neural network layers, such as convolutional layers, pooling layers, and fully connected layers. Users can use DREW to build a complete CNN neural network, or combine DREW with other existing neural network libraries, so as to achieve an efficient CNN inference process. Third, we present our DREW pipeline in a way that is compatible with C/C++, Java, and Python. Specifically, we provide sequential and parallel versions of the convolutional layer in C/C++.

VII. Evaluation

A. Experimental Setup

Methodology. We compare DREW with the original Winograd convolution without deep reuse [18], [38] in both serial and parallel modes. We first apply our approach to only a single convolutional layer to measure the single-layer speedups (Section VII-B). Second, we apply our approach to the full neural networks with the optimal clustering configurations from the single-layer experiments to measure the end-to-end speedups (Section VII-C). Third, we analyze the influence of different factors on performance, including the clustering configurations of clustering granularity L_{cb} and the number of hash functions $\hat{H}$, and the experiment configurations of the batch size and the number of threads (Section VII-E). Fourth, we analyze the runtime overhead of each part of our approach (Section VII-F).

Platforms. We conduct experiments to measure the performance of DREW on four platforms. First, we evaluate DREW on the edge device of Raspberry Pi 4 B with 8 GB LPDDR4 memory, as introduced in Section II-A. Second, we evaluate DREW on NVIDIA Jetson AGX Xavier, which is a powerful edge device equipped with an 8-core ARM v8.2 64-bit CPU and a 512-core Volta GPU, along with 32 GB LPDDR4 memory. Third, we measure DREW on a platform equipped with an Intel i7-7700 K CPU with 64 GB DDR4 memory. Fourth, we use another platform equipped with an Intel i9-9900 K with 64 GB DDR4 memory.

Workloads. We evaluate DREW with five different networks: LeNet-5 [25], CifarNet [26], VGG-11 [2], VGG-16 [2], and SqueezeNet [27], which are classic and have been evaluated in many works [23], [39], [40], [41], [42]. SqueezeNet is particularly useful for edge devices which is resource-constrained. It has been widely used in various computer vision applications. The dataset of LeNet-5 is MNIST [43] with sparse images of size 28×28. The dataset of CifarNet is CIFAR10 [37] with images of size 32×32. We modify the size of filters in LeNet-5 and CifarNet to 3×3 for our study and there is no influence on accuracy. The dataset of VGG-11, VGG-16 and SqueezeNet is ImageNet [44] with image size of 224×224. VGG-11 and VGG-16 use 3×3 filters exclusively in the convolution layers where the Winograd algorithm can be directly applied. SqueezeNet uses a 3×3 convolution in each of its $fire$ modules. For single-layer performance, we only report the performance results of the convolutional layers with 3×3 filters in SqueezeNet ($Firex\_e3$). For end-to-end performance, we report the performance results for the full SqueezeNet network, where only the 3×3 convolutional layers incorporate our work.

B. Single-Layer Performance

We perform experiments with a range of different clustering configurations and collect the speedup and accuracy results for the single convolutional layer of the networks.

Configuration. We first explore the clustering granularity L_{cb}, which represents the number of channels of tiles for clustering. We test the channel number of tiles of the factors of C, which is less than 4 in each convolutional layer. Note that we do not explore the L_{cb} value that is larger than 4 because it will cause a large remaining ratio and unendurable accuracy loss. For the number of hash functions $\hat{H}$, we explore a range from 10 to 30. In the following experiments, on Raspberry Pi, the number of threads is set to 4 in parallel mode. On Jetson AGX Xavier platform, the number of threads is set to 8 in parallel mode. On

TABLE II
SINGLE-LAYER PERFORMANCE BENEFITS. L_{cb} IS THE NUMBER OF CHANNELS OF TILES, $\hat{H}$ IS THE NUMBER OF HASH FUNCTIONS, r_c IS THE REMAINING RATIO, $Serial$ MEANS SERIAL SPEEDUP, $Parallel$ MEANS PARALLEL SPEEDUP, "CORE I7" AND "CORE I9" ARE $Core\ i7$ AND $Core\ i9$ PLATFORMS, "RASPBERRY PI" REPRESENTS THE RASPBERRY PI PLATFORM, "JETSON" REPRESENTS THE NVIDIA JETSON AGX XAVIER PLATFORM, AND Δ ACC IS THE ACCURACY LOSS

Network	Layer	Configuration		r_c	Serial				Parallel				Δ Acc
		L_{cb}	$\hat{H}$		Core i7	Core i9	Raspberry Pi	Jetson	Core i7	Core i9	Raspberry Pi	Jetson	
LeNet-5	Conv1	1	16	0.04	4.71×	4.51×	5.62×	2.26×	1.16×	1.11×	1.48×	1.13×	0.0008
	Conv2	1	12	0.14	5.05×	4.50×	6.01×	3.77×	1.42×	1.27×	1.52×	1.36×	0.0012
Average					4.88×	4.51×	5.81×	3.01×	1.29×	1.19×	1.50×	1.25×	0.0010
CifarNet	Conv1	1	16	0.14	6.52×	6.21×	5.37×	6.90×	1.77×	1.69×	1.39×	1.82×	0.0031
	Conv2	1	12	0.13	5.78×	5.67×	5.29×	5.17×	1.66×	1.63×	1.38×	1.35×	0.0027
Average					6.15×	5.94×	5.33×	6.03×	1.72×	1.66×	1.38×	1.59×	0.0029
VGG-11	Conv1	1	20	0.04	8.62×	8.50×	12.55×	7.78×	2.36×	2.32×	3.85×	2.48×	0.0087
	Conv2	1	23	0.07	8.59×	8.38×	11.17×	8.36×	2.54×	2.48×	3.39×	2.14×	0.0132
	Conv3-1	1	20	0.16	5.90×	6.08×	10.44×	8.97×	1.72×	1.77×	2.86×	2.19×	0.0071
	Conv3-2	1	17	0.06	9.17×	9.16×	6.08×	12.31×	2.75×	2.75×	1.73×	3.13×	0.0023
	Conv4-1	1	17	0.16	5.86×	5.90×	8.65×	7.98×	1.79×	1.80×	2.16×	2.03×	0.0030
	Conv4-2	1	15	0.09	6.67×	6.67×	8.20×	12.58×	2.05×	2.06×	2.07×	3.20×	0.0015
	Conv5-1	1	13	0.11	6.39×	6.55×	7.19×	8.27×	1.95×	2.00×	1.81×	2.11×	0.0013
	Conv5-2	1	11	0.08	7.28×	7.53×	7.70×	9.69×	2.23×	2.30×	1.93×	2.47×	0.0015
Average					7.31×	7.35×	8.99×	9.49×	2.17×	2.18×	2.48×	2.47×	0.0048
VGG-16	Conv1-1	1	24	0.05	7.67×	7.21×	10.88×	7.97×	2.23×	2.09×	2.74×	1.97×	0.0057
	Conv1-2	1	23	0.02	10.97×	9.18×	10.84×	6.87×	3.27×	2.74×	2.66×	1.73×	0.0075
	Conv2-1	1	22	0.09	8.40×	6.88×	10.66×	8.21×	2.76×	2.26×	2.78×	2.06×	0.0050
	Conv2-2	1	20	0.07	9.00×	8.29×	11.77×	9.11×	2.70×	2.49×	3.86×	2.29×	0.0018
	Conv3-1	1	21	0.20	5.08×	4.97×	8.79×	9.18×	1.64×	1.61×	2.26×	2.26×	0.0037
	Conv3-2	1	18	0.07	8.51×	7.94×	8.29×	11.23×	2.74×	2.55×	2.42×	2.86×	0.0012
	Conv3-3	1	16	0.06	9.19×	8.57×	8.84×	12.91×	2.96×	2.76×	2.59×	3.27×	0.0011
	Conv4-1	1	17	0.16	5.92×	5.70×	7.33×	9.01×	1.90×	1.83×	1.83×	2.34×	0.0019
	Conv4-2	1	15	0.08	6.77×	6.43×	7.59×	13.25×	2.22×	2.11×	1.92×	3.40×	0.0010
	Conv4-3	1	17	0.09	6.60×	6.35×	7.32×	10.86×	2.17×	2.09×	1.85×	2.77×	0.0006
	Conv5-1	1	16	0.20	4.71×	4.56×	7.45×	7.14×	1.54×	1.49×	1.80×	1.91×	0.0002
	Conv5-2	1	16	0.18	4.98×	5.03×	7.20×	7.79×	1.63×	1.65X	1.82×	1.99×	0.0005
	Conv5-3	1	11	0.07	7.68×	7.53×	8.36×	9.31×	2.52×	2.47×	2.02×	2.37×	0.0009
Average					7.35×	6.82×	8.87×	9.45×	2.33×	2.16×	2.35×	2.40×	0.0024
SqueezeNet	Fire2_e3	1	11	0.06	8.47×	8.34×	13.40×	8.33×	2.72×	2.68×	4.30×	2.20×	0.0021
	Fire3_e3	1	10	0.04	8.20×	8.01×	14.07×	8.86×	2.67×	2.61×	4.36×	2.25×	0.0037
	Fire4_e3	1	19	0.11	7.78×	8.01×	10.75×	5.87×	2.24×	2.30×	2.84×	1.53×	0.0084
	Fire5_e3	1	21	0.14	5.05×	5.04×	5.48×	4.47×	1.63×	1.63×	1.74×	1.14×	0.0069
	Fire6_e3	1	17	0.11	8.51×	8.57×	8.49×	5.66×	2.59×	2.61×	2.24×	1.44×	0.0067
	Fire7_e3	1	19	0.16	8.46×	8.48×	7.07×	4.49×	2.52×	2.53×	1.87×	1.16×	0.0063
	Fire8_e3	1	17	0.11	9.19×	9.42×	5.64×	4.73×	2.68×	2.74×	1.39×	1.22×	0.0081
	Fire9_e3	1	13	0.14	6.37×	6.58×	7.23×	4.97×	2.08×	2.14×	1.91×	1.28×	0.0043
Average					7.75×	7.81×	9.02×	5.92×	2.39×	2.41×	2.62×	1.53×	0.0058

Core i7 platform, the number is set to 8. On Core i9 platform, it is set to 16. We set the number of threads according to the maximum number of threads of the platform. Besides, on Jetson AGX Xavier, Core i7 platform and Core i9 platform, we fix the batch size to 100 for all neural networks. On the the Raspberry Pi platform, due to the memory limitations, we fix the batch size to 100 for LeNet-5, CifarNet and SqueezeNet, to 64 for VGG-11, and to 16 for VGG-16, which are common cases in real-life applications.

Performance-Accuracy Balance. We aim to achieve a balance between performance and accuracy. To measure the balance between performance improvement and accuracy loss, we define efficiency score: $e = -speedup \cdot log(accuracy\ loss)$. The higher value of e means that we obtain relatively higher performance with less accuracy loss.

Result. We report in Table II the speedups achieved by the configuration that has the highest efficiency score in each convolutional layer, and we have the following observations.

First, our approach leads to significant performance benefits. For the serial Winograd convolution, our approach delivers an average speedup of 8.53× on Raspberry Pi platform, 8.01× on Jetson AGX Xavier platform, 7.21× on Core i7 platform, and 6.99× on Core i9 platform. Even for the highly parallel implementation, DREW still provides an average speedup of 2.33× on Raspberry Pi platform, 2.09× on Jetson AGX Xavier platform, 2.21× on Core i7 platform, and 2.14× on Core i9 platform. The performance speedup is up to 14.07× and 4.36× in serial mode and parallel mode respectively, which proves the effectiveness of DREW.

Second, an appropriate combination of L_{cb} and $\hat{H}$ incurs a small remaining ratio, thus achieving significant time savings. Experiments show that our approach performs well when L_{cb} is one. Note that although larger L_{cb} leads to higher speedups,

TABLE III
END-TO-END PERFORMANCE BENEFITS. Δ ACC IS THE ACCURACY LOSS

Network	Serial speedup				Parallel speedup				Δ Acc
	Core i7	Core i9	Raspberry Pi	Jetson	Core i7	Core i9	Raspberry Pi	Jetson	
LeNet-5	4.39×	4.28×	4.51×	2.88×	1.18×	1.15×	1.18×	1.16×	0.0026
CifarNet	4.78×	4.34×	4.48×	4.24×	1.41×	1.36×	1.16×	1.18×	0.0064
VGG-11	7.43×	7.76×	7.42×	10.91×	2.21×	2.29×	1.96×	2.75×	0.0277
VGG-16	9.22×	8.51×	9.23×	5.68×	2.76×	2.41×	2.50×	1.43×	0.0123
SqueezeNet	4.83×	4.84×	5.37×	4.72×	1.42×	1.42×	1.49×	1.16×	0.0265

the accuracy loss could also be large, which results in a poor efficiency score (detailed in Section VII-E).

Third, on the networks whose image size is large, such as VGG-11 and VGG-16, the remaining ratio becomes smaller, and thus the achieved speedup is higher. Note that the first subconvolutional layer in each convolutional layer in VGG-11 and VGG-16, such as Conv3-1, Conv4-1, and Conv5-1, can have a larger remaining ratio compared to the other layers, because of the previous maximum pooling operation.

Accuracy Loss. Accuracy is another concern in lossy CNN inference. We report the single-layer accuracy loss of DREW compared to [23] in Table II, which implies that DREW incurs less than 1.32% accuracy loss in each layer. Note that we have a fine-tuning process to further reduce the accuracy loss, which is detailed in Section VII-D.

C. End-to-End Performance

To measure the end-to-end performance benefits of the full neural networks, we apply DREW to each convolutional layer with the configuration that can attain the best efficiency score mentioned in Section VII-B.

We show our performance benefits on the end-to-end execution time of the full neural networks in Table III. We involve all runtime overhead (clustering and others) in our time measurement and have the following observations.

First, our deep-reuse Winograd convolution achieves significant performance benefits. For the serial Winograd convolution, DREW achieves an average speedup of 6.20× on the Raspberry Pi platform, 5.69× on the Jetson AGX Xavier platform, 6.13× on Core i7 platform, and 5.95× on Core i9 platform. For the parallel implementation, it still provides an average speedup of 1.66× on Raspberry Pi platform, 1.54× on Jetson AGX Xavier platform, 1.80× on Core i7 platform, and 1.73× on Core i9 platform. The performance speedup is up to 9.23× and 2.76× in serial mode and parallel mode respectively.

Second, for LeNet-5, CifarNet and SqueezeNet, the end-to-end speedups are not as much as the speedups of single layers. The reason is that there are other layers, such as the activation layer and the pooling layer, mixed into CNNs, and Squeezenet also has some convolutional layers with a filter size other than 3×3 that cannot be accelerated using winograd convolution. In contrast, for VGG-11 and VGG-16, the end-to-end speedups are larger than the speedups of single layers, and the reason is that the remaining ratio is reduced for the subconvolutional layers, such as Conv3-2, Conv4-2, and Conv5-2, before pooling layers.

Third, the performance benefit among the end-to-end process is not as much as the computation saving because deep reuse introduces extra operations: clustering and the reconstruction of the feature maps after deep reuse optimization. As to the single-layer performance in Table II, we only need to perform 2% to 20% of the original computations with DREW in these networks, as indicated by the r_c column.

In addition, the accuracy loss is less than 2.77%, as shown in Table III. A fine-tuning process to further reduce the accuracy loss is introduced in Section VII-D.

D. Trade-Off Between Accuracy and Efficiency

In our experiment, DREW incurs an additional 0.26% to 2.77% accuracy loss if the model weights remain unchanged, due to the limitation that we cannot adjust the length of neuron vector to be less than 16. Since accuracy is important in many applications, we have the following designs to remain the accuracy in an acceptable range.

First, the network can be fine-tuned to amortize the accuracy loss, which is similar to prior practices [45]. We re-train the model using DREW. Therefore, only the model weights are updated, instead of changing the model architecture. After fine-tuning, DREW causes less than 0.4% accuracy loss on all networks.

Note that fine-tuning, as part of the training process, does not lengthen the inference time, so the reported inference speedups still remain.

Second, in our analysis, the first layer in CifarNet and the first three layers in VGG-11 and VGG-16 cause the major accuracy loss. We can apply deep reuse in the other layers while remaining the front layers unchanged.

Third, the balance between performance and accuracy loss can be adjusted to meet various application scenarios. The number of channels L_{cb} and the number of hash functions $\hat{H}$, mentioned in Section V-D, can be adjusted by users. For example, in a rigid scenario, users could set a small L_{cb} with a large $\hat{H}$ so that the accuracy loss shall be greatly minimized. We use LeNet-5 with 100 batch size and eight threads on Core i7 platform for illustration, and show the relation between accuracy loss and performance exchange of DREW in Fig. 9. The performance exchange is defined as the ratio difference between the performance of each measurement and the highest performance achieved.

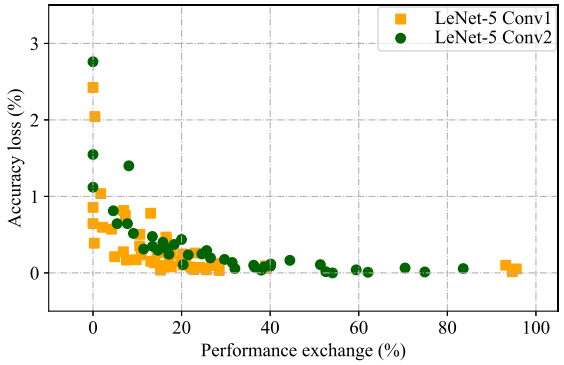


Fig. 9. Trade-off between accuracy and performance.

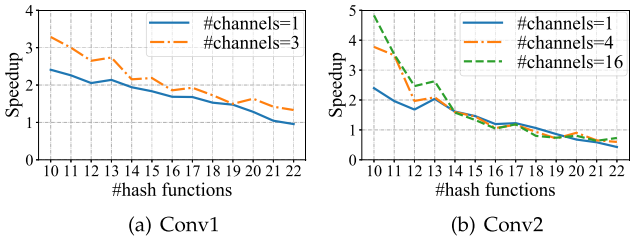


Fig. 10. Influence from different clustering configurations.

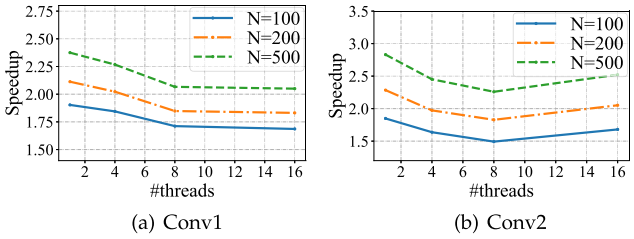


Fig. 11. Influence from batch size and the number of threads. N represents the batch size.

Fig. 9 shows that we can trade performance for accuracy in DREW.

E. Configuration Influence Analysis

To further analyze the efficacy of DREW, we explore the influence of different configurations on DREW and use CifarNet on Core i9 platform for illustration. For clustering granularity and hash size, we fix the batch size to 100 and the number of threads on CPUs to 16. When we analyze the batch size and thread number, we only vary the batch size and the number of threads, while retaining the other configurations unchanged. The performance results are shown in Figs. 10 and 11.

Hash Size. The number of hash functions mainly influences the remaining ratio (introduced in Section II-C). With the decreasing number of hash functions, more computation is saved due to a smaller remaining ratio. Therefore, the performance decreases along with the increase of the hash size, as shown in both Conv1 and Conv2 of Fig. 10.

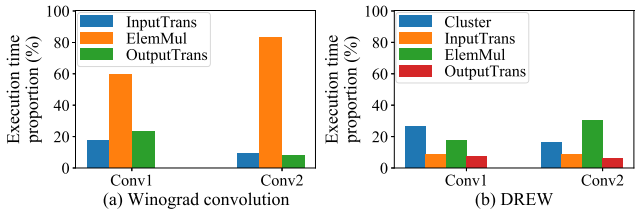


Fig. 12. Execution time analysis of Winograd convolution and DREW. $InputTrans$ represents input transformation, $ElemMul$ represents element-wise multiplication, $OutputTrans$ represents output transformation, and $cluster$ represents the clustering step of DREW.

Clustering Granularity. To study how clustering granularity affects the performance, we explore the channel number of tiles L_{cb} of all the factors of C in both convolutional layers. Fig. 10 shows that performance increases significantly as granularity becomes larger and the number of hash functions becomes smaller. The reason is that when the number of hash functions is small, the remaining ratio becomes small; at the same time, large granularity results in fewer addition operations in the output accumulation, which causes discontinuous memory accesses. However, with the increasing number of hash functions $\hat{H}$, the remaining ratio becomes large especially when the granularity is large, resulting in moderate performance benefits.

Batch Size. The remaining ratio becomes smaller as the batch size increases, introducing more performance benefits. For example, we explore the batch sizes of 100, 200, and 500 in Fig. 11. DREW achieves the highest speedups for both Conv1 and Conv2 when the batch size is 500.

Number of Threads. DREW achieves significant performance benefits in all cases. We explore the number of threads of 1, 4, 8, and 16 while keeping the other configurations unchanged in Fig. 11. For example, when the number of threads is 16, the speedup can be up to 2.52×. Note that the baseline and DREW use the same number of threads. The reason for increasing speedup with decreasing number of threads is that adding the number of threads increases the proportion of the extra overhead introduced by parallelism that cannot be amortized.

F. Execution Time Analysis

We analyze the execution time for each step of Winograd convolution and DREW in each layer, and use CifarNet on Core i9 platform for illustration. The results of the other benchmarks are similar. Fig. 12 shows the execution time proportions of each step of Winograd and DREW respectively.

For Winograd convolution, the proportions of input transformation, element-wise multiplication, and output transformation are 18%, 59%, and 23% for $Conv1$, and are 9%, 83%, and 8% for $Conv2$, respectively. For DREW, the proportions of clustering, input transformation, element-wise multiplication, and output transformation are 26%, 8%, 17%, and 7% for $Conv1$, and are 16%, 9%, 30%, and 6% for $Conv2$, respectively. Note that the results of DREW are the proportions of the execution time of each step of DREW to the total execution time of Winograd

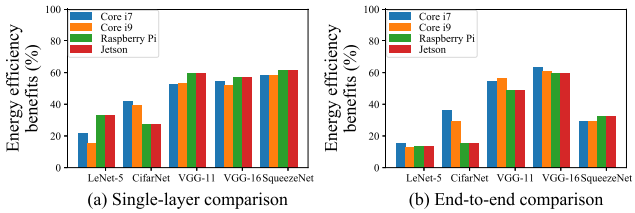


Fig. 13. Energy efficiency comparison between DREW and Winograd convolution on different platforms.

convolution. We have the following findings. First, the experimental results show that element-wise multiplication is the most time-consuming step in Winograd convolution. Second, our work introduces some clustering overhead, but significantly reduces the runtime of input transformation, element-wise multiplication, and output transformation in Winograd convolution.

G. Energy Efficiency

Energy is an important consideration for modern processors, especially for the edge device executing inferences. DREW greatly reduces energy consumptions by reducing the number of computations. Similar to [46], we use performance per Watt as the metric to measure energy efficiency. The Thermal Design Power (TDP) is 6.25 W on Raspberry Pi 4 Model B, 30 W on NVIDIA Jetson AGX Xavier, 95 W on i9-9900 K, and 91 W on i7-7700 K. We show the energy efficiency comparison between DREW and the original Winograd convolution on different platforms in parallel mode in Fig. 13, including the single-layer and end-to-end performance comparison.

In detail, DREW brings 45% extra energy savings in single-layer comparison, and 35% extra energy savings in end-to-end comparison. For networks such as LeNet-5, CifarNet and SqueezeNet, the overall end-to-end energy efficiency benefit is not as much as that of single layers. The reason is that extra layers such as pooling, fully-connected layers, and convolutional layers of other sizes exist in end-to-end models, which also consume resources.

H. Applicability to Other Networks

DREW can be applied to other convolutional neural networks. We select LeNet-5, CifarNet, VGG-11, VGG-16 and SqueezeNet for validation because they are classic and have been evaluated in many works. More advanced models have not substantially changed the use of convolution. For example, the Conv1-2 of VGG-16 [2], which has 64 input features and 64 output features, is the same as Conv2_1 in ResNet-18 [1]. Please note that the Winograd algorithm has the limitation that it can only be applied to convolutions with 3×3 filters. Because DREW is built on Winograd, the filters of convolutional layers should be presented as 3×3. Furthermore, advanced models only make the weights more efficient without considering the input similarities. Therefore, our work still can be used in other situations.

VIII. Related Work

In this section, we introduce the related work from data reuse, Winograd, and compression perspectives.

Data Reuse. Data reuse has been proved to be a great success in data management and analytics [47], [48], [49], [50], [51], [52]. Similarity and redundancy are utilized to reduce the amount of computation and space. Zhang et al. [48] developed text analytics directly on compression (TADOC) in which the key idea is data reuse: for a segment of duplicated content, TADOC only stores it once to save space and computes it once to reduce computation. DeepSqueeze [51] added semantic compression technology to traditional data compression algorithms to recognize the correlation between tabular data columns for reusing. Li et al. [52] compressed the uncertain trajectory data in Road Networks by mining and reusing the similarity between trajectories. Data reuse has also been applied to deep convolutional neural networks, which is called deep reuse. Deep reuse is an acceleration method that speedups convolution by reusing the similarities among neuron vectors. The closest work to DREW is [23], which applies the deep reuse in CNN inference process. Ning et al. [10] also applied the deep reuse in CNN training process on the fly. Different from these works, we combine deep reuse with Winograd together, which further improves the performance of CNN.

Winograd. Cook [53] and Toom [54] proposed a class of minimal filtering algorithms, and Winograd [34] generalized these fast CNN algorithms. Lavin [18] further extended Winograd as an efficient convolution operation. There are many works on accelerating the Winograd algorithm. Some works are devoted to solving the limitations of the Winograd algorithm. To tackle the problem that Winograd is only effective on convolutions with kernel size as 3×3 and stride as 1, Jiang [55] proposed a nested Winograd algorithm, which uses an iterative decomposition method to turn the large convolution kernel into a series of 3×3 tiles. Yepez [56] extended the Winograd algorithm to a stride of 2, which is also valid for one, two, or three dimensions. Decomposable Winograd Method (DWM) [57] breaks through the limitation of the original Winograd's minimal filtering algorithm to wide and general convolutions. It decomposes kernels with large size or large stride to several small kernels with stride as 1 for further applying Winograd methods, so as to reduce the number of multiplications while keeping the numerical accuracy. There are studies applying Winograd to further enhance the acceleration of CNN inference. For example, Li et al. [58] integrated model compression quantization technology to represent the original floating-point value as a low-precision code, thereby gaining potentials for acceleration. Other works focus on speeding up Winograd convolution on specific hardware. Jia et al. [21] optimized the Winograd-based convolution on Intel Xeon Phi platforms. Yan et al. [22] optimized the batched Winograd algorithm on GPUs. Moreover, the Winograd algorithm has been integrated into many popular libraries, such as FALCON [59], LIBXSMM [60], MKL-DNN [20], and SASS [61].

Model Compression. Model compression techniques have been proposed to save the computation time and memory occupation of CNNs [62], [63], [64], [65], [66], [67]. Model

compression can be divided into four categories [68]: pruning and quantization, low-rank factorization, transferred/compact convolutional filters, and knowledge distillation. Deep reuse, which leverages the similarities among input online, is orthogonal to model compression techniques, which reduces the model size offline by leveraging the similarities among weights. Deep reuse can be applied to compressed models, as discussed in [23], and hence they are complementary to each other.

In general, we extend the idea of data reuse to Winograd, and provide a new application scenario for applying data engineering technologies to deep learning at the edge. We believe that our work can shed lights on the future research of applying CNN inference to edge devices.

IX. Conclusion

This article combines deep reuse with Winograd convolution, and yields a library, called DREW, to enable efficient CNN inference at the edge. By enabling deep reuse in the Winograd algorithm, DREW reduces the number of convolution operations to an average of 10% of the original operations. The paper presents how deep reuse can be applied to Winograd, thus enabling efficient CNN inference on edge device. It discusses the major challenges when applying deep reuse to Winograd convolution, and provides a set of solutions in applying our method. In evaluation, DREW provides 7.69× performance improvement over the original Winograd convolution with almost no accuracy loss, and achieves up to 62% energy efficiency benefits.

References

[1] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 770–778.

[2] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," 2014, *arXiv:1409.1556*.

[3] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2017, pp. 2261–2269.

[4] A. Ordookhanians et al., "Demonstration of krypton: Optimized cnn inference for occlusion-based deep CNN explanations," *Proc. VLDB Endow.*, vol. 12, no. 12, pp. 1894–1897, 2019.

[5] D. Kang et al., "NoScope: Optimizing neural network queries over video at scale," *Proc. VLDB Endow.*, vol. 10, pp. 1586–1597, 2017.

[6] S. Nakandala, A. Kumar, and Y. Papakonstantinou, "Incremental and approximate inference for faster occlusion-based deep CNN explanations," in *Proc. Int. Conf. Manage. Data*, New York, NY, USA: Association for Computing Machinery, 2019, pp. 1589–1606.

[7] J. Chen and X. Ran, "Deep learning with edge computing: A review," *Proc. IEEE Proc.*, vol. 107, no. 8, pp. 1655–1674, Aug. 2019.

[8] N. Band, "MemFlow: Memory-aware distributed deep learning," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2020, pp. 2883–2885.

[9] Q. Xu et al., "Payment behavior prediction on shared parking lots with TR-GCN," *The VLDB J.*, vol. 31, pp. 1–24, 2022.

[10] L. Ning, H. Guan, and X. Shen, "Adaptive Deep Reuse: Accelerating CNN Training on the Fly," in *Proc. Int. Conf. Data Eng.*, 2019, pp. 1538–1549.

[11] P. Bellini, P. Nesi, and G. Pantaleo, "IoT-enabled smart cities: A review of concepts, frameworks and key technologies," *Appl. Sci.*, vol. 12, no. 3, 2022, Art. no. 1607.

[12] B. Jacob et al., "Quantization and training of neural networks for efficient integer-arithmetic-only inference," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2018, pp. 2704–2713.

[13] Y. Liu et al., "Optimizing CNN model inference on CPUs," in *Proc. USENIX Annu. Techn. Conf.*, 2019, pp. 1025–1039.

[14] S. Nakandala and A. Kumar, "Vista: Optimized system for declarative feature transfer from deep cnns at scale," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2020, pp. 1685–1700.

[15] A. Ordookhanians et al., "Demonstration of krypton: Optimized CNN inference for occlusion-based deep CNN explanations," *Proc. VLDB Endowment*, vol. 12, pp. 1894–1897, 2019.

[16] A. G. Howard et al., "MobileNets: Efficient convolutional neural networks for mobile vision applications," 2017, *arXiv:1704.04861*.

[17] M. Tan et al., "MnasNet: Platform-aware neural architecture search for mobile," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2019, pp. 2815–2823.

[18] A. Lavin and S. Gray, "Fast algorithms for convolutional neural networks," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 4013–4021.

[19] cuDNN: Efficient primitives for deep learning, 2014. [Online]. Available: https://developer.nvidia.com/cudnn

[20] Intel(R) math kernel library for deep neural networks, 2016. [Online]. Available: https://github.com/oneapi-src/oneDNN

[21] Z. Jia et al., "Optimizing N-dimensional, winograd-based convolution for manycore CPUs," in *Proc. 25th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2018, pp. 109–123.

[22] D. Yan, W. Wang, and X. Chu, "Optimizing batched winograd convolution on GPUs," in *Proc. 25th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2020, pp. 32–44.

[23] L. Ning and X. Shen, "Deep reuse: Streamline CNN inference on the fly via coarse-grained computation reuse," in *Proc. ACM Int. Conf. Supercomputing*, 2019, pp. 438–448.

[24] R. Wu et al., "DREW: Efficient winograd CNN inference with deep reuse," in *Proc. ACM Web Conf.*, 2022, pp. 1807–1816.

[25] Y. Lecun et al., "Comparison of learning algorithms for handwritten digit recognition," in *Proc. Int. Conf. Artif. Neural Netw.*, 1995, pp. 53–60.

[26] "CifarNet," 2020. [Online]. Available: http://places.csail.mit.edu/deepscene/small-projects/TRN-pytorch-pose/model_zoo/models/slim/nets/

[27] F. N. Iandola et al., "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <1mb model size," 2016, *arXiv:1602.07360*.

[28] "Raspberry Pi 4 model B," 2022. [Online]. Available: https://www.raspberrypi.com

[29] M. D. Mudaliar and N. Sivakumar, "IoT based real time energy monitoring system using raspberry Pi," *Internet Things*, vol. 12, 2020, Art. no. 100292.

[30] G. Bekaroo and A. Santokhee, "Power consumption of the raspberry Pi: A comparative analysis," in *Proc. IEEE Int. Conf. Emerg. Technol. Innov. Bus. Pract. Transformation Societies*, 2016, pp. 361–366.

[31] C. W. Zhao, J. Jegatheesan, and S. C. Loon, "Exploring IoT application using raspberry Pi," *Int. J. Comput. Netw. Appl.*, vol. 2, no. 1, pp. 27–34, 2015.

[32] S. S. Prabha, A. J. P. Antony, M. J. Meena, and S. R. Pandian, "Smart cloud robot using raspberry pi," in *Proc. IEEE Int. Conf. Recent Trends Inf. Technol.*, 2014, pp. 1–5.

[33] Y. Sun, L. Geng, and K. Dan, "Design of smart mirror based on raspberry pi," in *Proc. IEEE Int. Conf. Intell. Transp. Big Data Smart City*, 2018, pp. 77–80.

[34] S. Winograd, "Arithmetic complexity of computations," *Siam*, 1980.

[35] A. Gionis, P. Indyk, and R. Motwani, "Similarity search in high dimensions via hashing," in *Proc. 25th Int. Conf. Very Large Data Bases*, 1999, pp. 518–529.

[36] Y.-C. Li, C.-M. Yeh, and C.-C. Chang, "Data hiding based on the similarity between neighboring pixels with reversibility," *Digit. Signal Process.*, vol. 20, no. 4, pp. 1116–1128, 2010.

[37] A. Krizhevsky and G. Hinton, "Learning multiple layers of features from tiny images," 2009.

[38] C. Zhang, F. Zhang, X. Guo, B. He, X. Zhang, and X. Du, "iMLBench: A machine learning benchmark suite for CPU-GPU integrated architectures," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 7, pp. 1740–1752, Jul. 2021.

[39] W. Wen et al., "Learning structured sparsity in deep neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, 2016, pp. 2082–2090.

[40] H. Wang et al., "ATOMO: Communication-efficient learning via atomic sparsification," in *Proc. Adv. Neural Inf. Process. Syst.*, 2018, pp. 9872–9883.

[41] X. Zhang, J. Zou, K. He, and J. Sun, "Accelerating very deep convolutional networks for classification and detection," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 38, no. 10, pp. 1943–1955, Oct. 2016.

[42] J. Tang et al., "Enabling deep learning on IoT devices," *Computer*, vol. 50, no. 10, pp. 92–96, Oct. 2017.

[43] Y. LeCun, C. Cortes, and C. J. Burges, "THE mnist database of handwritten digits," 1998. [Online]. Available: http://yann.lecun.com/exdb/mnist/

[44] O. Russakovsky et al., "ImageNet large scale visual recognition challenge," *Int. J. Comput. Vis.*, vol. 115, pp. 211–252, 2015.

[45] M. Figurnov et al., "PerforatedCNNs: Acceleration through elimination of redundant convolutions," in *Proc. Adv. Neural Inf. Process. Syst.*, 2016, pp. 1–9.

[46] K. Zhang, J. Hu, B. He, and B. Hua, "DIDO: Dynamic pipelines for in-memory key-value stores on coupled CPU-GPU architectures," in *Proc. IEEE 33rd Int. Conf. Data Eng.*, 2017, pp. 671–682.
[47] F. Zhang et al., "Zwift: A programming framework for high performance text analytics on compressed data," in *Proc. ACM Int. Conf. Supercomput.*, 2018, pp. 195–206.
[48] F. Zhang et al., "Efficient document analytics on compressed data: Method, challenges, algorithms, insights," *Proc. VLDB Endowment*, vol. 11, no. 11, pp. 1522–1535, 2018.
[49] F. Zhang, J. Zhai, X. Shen, O. Mutlu, and X. Du, "Enabling efficient random access to hierarchically-compressed data," in *Proc. IEEE 36th Int. Conf. Data Eng.*, 2020, pp. 1069–1080.
[50] F. Zhang et al., "TADOC: Text analytics directly on compression," *VLDB J.*, vol. 30, pp. 163–188, 2021.
[51] A. Ilkhechi et al., "Deepsqueeze: Deep semantic compression for tabular data," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2020, pp. 1733–1746.
[52] T. Li et al., "Compression of uncertain trajectories in road networks," *Proc. VLDB Endowment*, vol. 13, no. 7, pp. 1050–1063, 2020.
[53] S. Cook, "On the minimum computation time for multiplication," Ph.D. dissertation, Harvard U., Cambridge, Mass, 1966.
[54] A. L. Toom, "The complexity of a scheme of functional elements realizing the multiplication of integers," Soviet Mathematics Doklady, vol. 3, no. 4, pp. 714–716, 1963.
[55] J. Jiang, X. Chen, and C.-Y. Tsui, "A reconfigurable winograd CNN accelerator with nesting decomposition algorithm for computing convolution with large filters," 2021, *arXiv:2102.13272*.
[56] J. Yepez and S.-B. Ko, "Stride 2 1-D, 2-D, and 3-D winograd for convolutional neural networks," *IEEE Trans. Very Large Scale Integration Syst.*, vol. 28, no. 4, pp. 853–863, Apr. 2020.
[57] D. Huang et al., "DWM: A decomposable winograd method for convolution acceleration," in *Proc. AAAI Conf. Artif. Intell.*, 2020, pp. 4174–4181.
[58] G. Li, L. Liu, X. Wang, X. Ma, and X. Feng, "Lance: Efficient low-precision quantized winograd convolution for neural networks based on graphics processing units," in *Proc. IEEE Int. Conf. Acoust. Speech Signal Process.*, 2020, pp. 3842–3846.
[59] "FALCON Library: Fast image convolution in neural networks on intel architecture," 2016, [Online]. Available: https://colfaxresearch.com/falcon-library/
[60] "LIBXSMM," 2020. [Online]. Available: https://github.com/hfp/libxsmm
[61] "MaxAs," 2020. [Online]. Available: https://github.com/NervanaSystems/maxas
[62] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," in *Proc. Int. Conf. Learn. Representations*, 2016, pp. 1–14.
[63] P. Molchanov et al., "Pruning convolutional neural networks for resource efficient inference," in *Proc. 5th Int. Conf. Learn. Representations*, Toulon, France, 2017, pp. 1–17.
[64] J. Wu, C. Leng, Y. Wang, Q. Hu, and J. Cheng, "Quantized convolutional neural networks for mobile devices," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 4820–4828.
[65] C. Zhu et al., "Trained ternary quantization," in *Proc. 5th Int. Conf. Learn. Representations*, Toulon, France, 2017, pp. 1–10.
[66] M. Rastegari et al., "XNOR-Net: Imagenet classification using binary convolutional neural networks," in *Proc. Eur. Conf. Comput. Vis.*, ser. Lecture Notes in Computer Science, B. Leibe and J. Matas Eds., Amsterdam, The Netherlands, Oct. 11–14, 2016, pp. 525–542.
[67] G. Hinton, O. Vinyals, and J. Dean, "Distilling the knowledge in a neural network," in *Proc. Int. Conf. Neural Inf. Process. Syst. Deep Learn. Representation Learn. Workshop*, 2015, pp. 1–9.
[68] Y. Cheng et al., "A survey of model compression and acceleration for deep neural networks," 2017, *arXiv:1710.09282*.

Feng Zhang (Member, IEEE) received the bachelor's degree from Xidian University, in 2012, and the PhD degree in computer science from Tsinghua University, in 2017. He is an associate professor with the Key Laboratory of Data Engineering and Knowledge Engineering (MOE), Renmin University of China. His major research interests include parallel and distributed systems.

Ruofan Wu received the bachelor's degree from the Renmin University of China, in 2021. She is currently working toward the master degree in computer science with the Renmin University of China, advised by Prof. Feng Zhang. Her current research interests include parallel computing, heterogeneous computing, and parallel accelerating.

Jiawei Guan received the bachelor's degree from the Renmin University of China, in 2022. She is currently working toward the PhD degree in computer science with the Renmin University of China, advised by prof. Feng Zhang. Her research interests include high performance computing and machine learning.

Zhen Zheng received the PhD degree from the Department of Computer Science and Technology, Tsinghua University, China, in 2019. He was a Visiting Scholar of North Carolina State University, in 2018. He joined Alibaba in August 2019 as a Researcher. His research interests include AI compiler, large scale machine learning systems, and heterogeneous computing.

Xiaoguang Guo received the bachelor's degree from the Renmin University of China, in 2022. He joined the Key Laboratory of Data Engineering and Knowledge Engineering (MOE), in 2020. His research interests include high performance computing, machine learning, and parallel and distributed systems.

Xiao Zhang received the master's degree from Renmin University and the PhD degree from the Institute of Computing Technology, Chinese Academy of Science, in 1998 and 2001, respectively, both in computer science and technology. He is a professor with the School of Information, Renmin University of China. His research interests include Big Data management systems.

Xiaoyong Du received the BS degree from Hangzhou University, Zhejiang, China, in 1983, the ME degree from the Renmin University of China, Beijing, China, in 1988, and the PhD degree from the Nagoya Institute of Technology, Nagoya, Japan, in 1997. He is currently a professor with the School of Information, Renmin University of China. His current research interests include databases and intelligent information retrieval.

Xipeng Shen (Member, IEEE) received the PhD degree in computer science from the University of Rochester, in 2006. He is a professor in Computer Science with the North Carolina State University. He is a receipt of the DOE Early Career Award, NSF CAREER Award, Google Faculty Research Award, and IBM CAS Faculty fellow Award. He is an ACM distinguished member, ACM distinguished speaker. His interest is in Programming Systems and Machine Learning.