DREW: Efficient Winograd CNN Inference with Deep Reuse

Ruofan Wu[◇], Feng Zhang[◇], Jiawei Guan[◇], Zhen Zheng[★], Xiaoyong Du[◇], Xipeng Shen[+]
[◇]Key Laboratory of Data Engineering and Knowledge Engineering (MOE),
and School of Information, Renmin University of China
[★]Alibaba Group
[+]Computer Science Department, North Carolina State University
ruofanwu@ruc.edu.cn,fengzhang@ruc.edu.cn,guanjw@ruc.edu.cn,james.zz@alibaba-inc.com,duyong@ruc.edu.cn,xshen5@ncsu.edu

ABSTRACT

Deep learning has been used in various domains, including Web services. Convolutional neural networks (CNNs), which are deep learning representatives, are among the most popular neural networks in Web systems. 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. The limited computing resource and high computation pressure limit the effective use of CNN algorithms in industry. Fortunately, a minimal filtering algorithm called Winograd can reduce convolution calculations by minimizing multiplication operations. We find that Winograd convolution can be sped up 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. Experiments show that 1) DREW further accelerates the Winograd convolution by an average of 2.06× speedup; 2) when DREW is applied to end-to-end Winograd CNN inference, it achieves 1.71× the average performance speedup with no (<0.4%) accuracy loss; 3) DREW reduces the number of convolution operations to 11% of the original operations on average.

CCS CONCEPTS

• **Information systems** → **World Wide Web**; • **Computing methodologies** → *Neural networks.*

KEYWORDS

data reuse, deep reuse, Winograd, Web systems

ACM Reference Format:
Ruofan Wu[◇], Feng Zhang[◇], Jiawei Guan[◇], Zhen Zheng[★], Xiaoyong Du[◇], Xipeng Shen[+]. 2022. DREW: Efficient Winograd CNN Inference with Deep Reuse. In *Proceedings of the ACM Web Conference 2022 (WWW '22), April 25–29, 2022, Virtual Event, Lyon, France.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3485447.3511985

WWW '22, April 25–29, 2022, Virtual Event, Lyon, France

ACM ISBN 978-1-4503-9096-5/22/04…$15.00
https://doi.org/10.1145/3485447.3511985

1 INTRODUCTION

Convolutional Neural Networks (CNNs) have shown successes and gained popularity in Web systems [8, 12, 27, 28, 30, 34, 40, 47, 51]. Different from the training process, inferences of CNN are widely processed on CPUs in industry and face a high demand for performance optimization [19, 29, 63]. Note that accelerators with high computing power are usually too expensive for companies to apply to inference workloads. In industry, the CNNs are usually trained on HPC clusters, while the inference could be conducted on a less powerful machine, such as CPUs or mobile processors [10, 16, 37, 41, 43, 55, 60]. Due to high compute density, it is important to optimize the inference process on CPUs, especially for industry usage. The key to improving the inference performance is to accelerate the convolutional layers of CNN models, which are computationally intensive and dominate the total execution time.

Winograd convolution [23] has been demonstrated to be a very efficient CNN optimization technique. Employing Winograd minimal filtering algorithm reduces the arithmetic complexity of convolution operations by at least 2.25× [23] theoretically, saving substantial time and energy. The majority of modern deep learning libraries, including Nvidia cuDNN [1] and Intel oneDNN (previously known as MKL-DNN) [3], enable Winograd convolution for CNNs . Additionally, several attempts have been made to accelerate Winograd convolution via increased hardware efficiency [20, 50].

Prior research on Winograd convolution concentrated on how to improve the algorithm's performance on certain hardware platforms, such as GPUs [1, 50], rather than the algorithm's structure. We find that, rather than performing typical code optimizations, a unique approach called *deep reuse* can uncover and leverage reused calculations to accelerate convolutions. This approach reuses intermediate results in CNN inference by recognizing similarities among neuron vectors, saving both space and time on the fly [32]. The present deep reuse approach, however, is limited to GEMM-based convolution. The performance of CNN inference could be considerably enhanced if deep reuse is applied to the Winograd convolution.

Applying *deep reuse* to Winograd convolution requires to handle the following three challenges.

- *Algorithm design.* Winograd convolution involves fixed minimal filtering patterns, so there is no direct neuron vector to extract in the Winograd algorithm. Consequently, an appropriate method

needs to be designed for exploiting the similarities and saving computations.

- *Introduced overhead. Deep reuse* is an on-line process in which the similarity detection process among neuron vectors happens during the inference time. Because of the computation savings in Winograd convolution, it poses a tighter constraint on the introduced overhead compared to the GEMM-based convolution.
- *Cost-benefit tradeoff.* The limitations of minimal filtering patterns also prevent us from adjusting the reuse granularity as flexibly as original *deep reuse* does. Since *deep reuse* is a lossy optimization, to provide the choice to trade-off between performance and accuracy loss, a novel method to adjust reuse granularity needs to be designed.

We present our solution, a new inference method, called **DREW**, for applying **d**eep **re**use to **W**inograd convolution on modern processors. 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.

We evaluate DREW on three popular neural networks: LeNet-5 [25], CifarNet [4], and VGG-16 [36]. For single-layer performance, DREW achieves 6.68× performance improvement on average compared to the Winograd convolution without deep reuse. Even for the highly parallel Winograd implementation, DREW can still provide 1.11× to 3.27× performance improvement. For end-to-end performance, DREW achieves 5.92× performance improvement with no (<0.4%) accuracy loss, and for parallel implementation, DREW still maintains 1.15× to 2.76× performance improvement. With detailed analysis, the convolution operations can be reduced to 4 to 20% of the original computations and take up only 35 to 56% of the execution time.

As far as we know, DREW is the first work that combines deep reuse with Winograd convolution. In summary, this work makes the following contributions:

- It points out that deep reuse can be efficiently combined with Winograd convolution for the first time. This work proposes a new inference method, called DREW, which can detect and exploit input similarities among Winograd minimal filtering computation patterns.
- It designs a novel clustering process for DREW, 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 demonstrate its significant performance benefits with almost no accuracy loss.

2 BACKGROUND

2.1 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 [46]. 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 [3, 20, 50].

Workflow. The workflow of the convolutional layer with $F(2 \times 2, 3 \times 3)$ Winograd minimal filtering algorithm is shown in Figure 1. 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 channels 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.

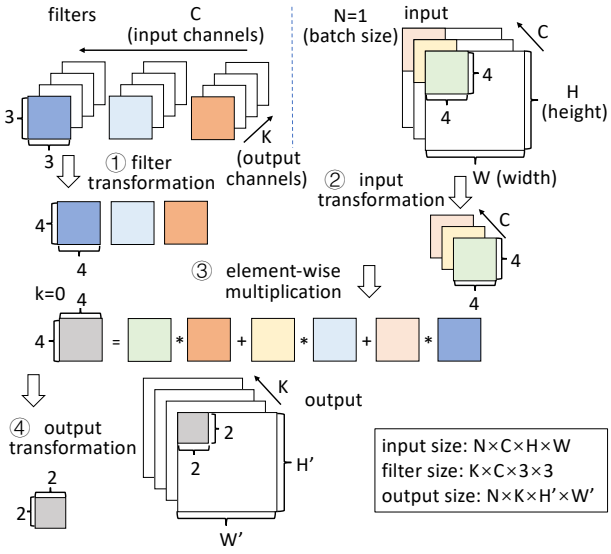


Figure 1: The workflow of Winograd convolution.

For online 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.

2.2 Deep Reuse

Deep reuse is a kind of optimization for accelerating CNN inferences by detecting and utilizing runtime similarities among input data [32]. 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 Figure 2 (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.

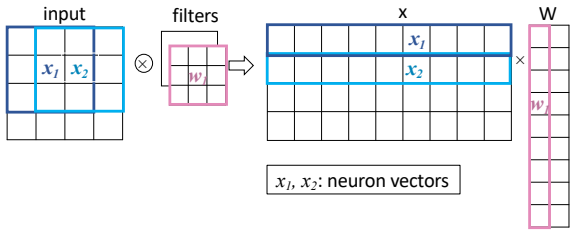


Figure 2: The illustration of GEMM-based convolution.

Clustering method. *Deep reuse* uses Locality Sensitive Hashing (LSH) [13] as the clustering method to detect similarities among neuron vectors because LSH can give 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 Equation 1:

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

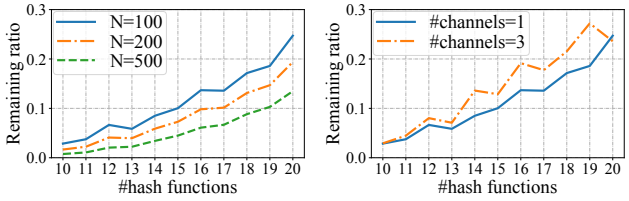
With H random vectors, LSH maps an input vector into a bit vector with 2^H possibilities. The input vectors that are close to each other have a high probability to be hashed into the same bit vector. Thus, the roughness of the clustering can be adjusted by the number of hash functions. After LSH being 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.* [32] define **remaining ratio** r_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.

3 MOTIVATION

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.

Opportunity. Based on our observation, neuron similarities exist in Winograd convolution. For Winograd-based convolution, the 4×4 tiles are local neurons as shown in Figure 1. Due to the continuity in images or feature maps that CNN often targets, it has been proved that neighbor neurons have extremely strong similarities [26]. Hence, there is a strong chance 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.

Observation. To prove our assumption on tile similarities, we conduct a series of experimental analysis and draw the conclusion that applying *deep reuse* to Winograd convolution has a tremendous potential for performance enhancement. We employ the CifarNet [4] trained model on CIFAR10 [22], and conduct its inference. We perform LSH with a variable number of hash functions to two convolutional layers and report the remaining ratio after clustering. Note that a greater number of hash functions results in a more precise clustering, and the remaining ratio indicates the reusable potentials, as discussed in Section 2.2. Figure 3 shows the experimental results of *Conv1* (*Conv2* exhibits similar trends). First, we cluster tiles within each channel with varying batch sizes and present the average remaining ratio for each channel in Figure 3 (a). The remaining ratio averages out at 0.084 and rises in tandem with the increase in hash size and decrease in batch size. Then, we cluster tiles from 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 Figure 3 (b). The findings demonstrate that tiles with multiple channels still create a tiny remaining ratio, especially when the number of hash functions is modest. A smaller remaining ratio is often associated with fewer channels. To sum up, large data similarities exist among tiles of Winograd convolution, which provides opportunities for reuse.



(a) Single channel. (b) Multiple channels.

Figure 3: Remaining ratio. (a) Different numbers of hash functions with varying batch sizes in a single channel. (b) Different numbers of hash functions with varying numbers of channels.

Importance. *Deep reuse* has been shown to significantly improve the performance of CNN inference [32]. Meanwhile, Winograd convolution results in a 2.25× reduction in multiplication calculations when compared to direct convolution [23]. It is worth the effort to further optimize due to the high performance gains. Furthermore, as previously explained, the tile in Winograd convolution is suited as the object we reuse. Since the experiments above have proved that the tiles of batched input have large similarity, utilizing the similarity allows for significant calculation savings, and it is possible to provide faster Winograd convolution.

4 SOLUTION OVERVIEW

We show in Section 3 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 1.

Idea. Revisiting the process of Winograd convolution mentioned in Section 2.1, 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.

Case study. We show an example in Figure 4 for processing a 1×2×6×6 input with 2×2×3×3 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 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.

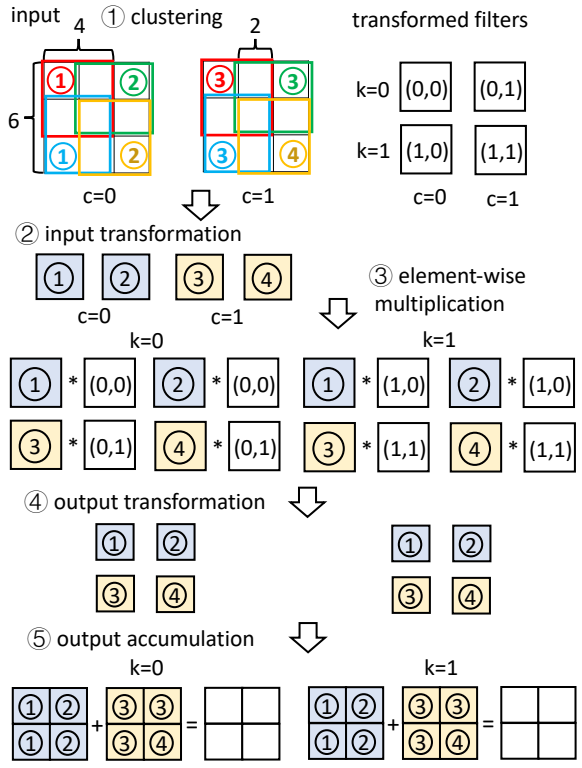


Figure 4: Example of applying deep reuse to Winograd.

Solutions to challenges. We develop DREW to solve the challenges listed in Section 1. First, we design a method to leverage the neuron similarities in Winograd convolution in which the tile size 16 is the smallest clustering granularity (Section 4 and Section 5.1). 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 5.2). 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 5.3).

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 5.1). Second, we provide novel clustering designs to reduce online cost within a reasonable range (Section 5.2). Third, we provide a novel adjustable method for users in clustering granularity balancing the trade-off between performance and accuracy (Section 5.3). Then, we develop a fine-tuning process to maintain high accuracy (Section 7.4).

5 DREW ALGORITHM AND OPTIMIZATIONS

After presenting the idea of combining deep reuse and Winograd convolution in Section 4, we introduce the detailed design of DREW in this section. First, we present the basic workflow of DREW. Then, we delve into the design of clustering and extend the clustering granularity to multiple channels.

5.1 Deep-Reuse Winograd

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 2.1), 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 Equation 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 Equation 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 Equation 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 Equation 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 Equation 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 Equation 7.

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

5.2 Clustering Design

We revisit our clustering design of *Step 1* mentioned in Section 5.1. Since Winograd is an online process, Winograd poses a tight constraint on clustering time, including 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. For our batched *deep-reuse Winograd* in DREW, instead of indexing the vectors one by one, we perform LSH projection by an H×16×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.

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 parallel in applying deep reuse in Winograd of DREW, because we need to record the buckets with their related vectors and calculate the bucket size.

Before showing our bucket mapping design, we first revisit the bucket mapping in [32], which treats the integer value of bit vector as cluster ID (to distinguish it from ours, we denote ours as bucket ID). They minimize 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×$2^{\hat{H}}$ neuron vectors for later element-wise multiplication. This would be a huge space overhead and would become a burden especially on the industrial hardwares where CNN inferences commonly run.

We develop a novel bucket mapping strategy in DREW. First, we iterate over P values in C channels. Second, we map each integer value of bit vector to buckets. Third, we increase the newly discovered bucket ID in order and count bucket size. In this way, we only need $O(\sum N_b)$ space for clustering and later computation. Because the integer values of vectors provide a hash-map with a $O(1)$ 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 7.6). Finally, we compute the cluster centroid of each bucket.

5.3 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 basic 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.

6 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.

Parallel design. All the five steps of DREW described in Section 5.1 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 $O(N{\cdot}H{\cdot}W{\cdot}C{\cdot}K)$ [23]. Assume that input channels are divided into N_{cb} channel blocks and each channel block contains tiles of L_{cb} channels (C=N_{cb} · L_{cb}).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 $O(C{\cdot}N{\cdot}H{\cdot}W{\cdot}\hat{H})$. 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 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 $O((\frac{\hat{H}}{K} + r_c + \frac{1}{L_{cb}}) \cdot N \cdot H \cdot W \cdot C \cdot K)$.

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 of *deep reuse* for 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 addition 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 ($\frac{\hat{H}}{K} + r_c + \frac{1}{L_{cb}}$), which is a coefficient in DREW's complexity, thus resulting in more computation savings.

Implementation. We build a library called DREW to integrate our work for three reasons. First, DREW can ease the burden on programmers to utilize deep reuse on Winograd algorithms; to this end, we present user-friendly APIs that is compatible with C/C++, Java, and Python. Second, DREW provides a highly efficient implementation of Winograd, which can be directly applied to many existing projects. Third, we hope that our integration of deep reuse and Winograd can be really integrated to other popular deep learning systems and libraries. Specifically, we develop Winograd in C/C++, and provide sequential and parallel versions. Moreover, we will open source our code after the paper being accepted.

7 EVALUATION

7.1 Experimental Setup

Methodology. We compare our DREW with the original Winograd convolution without deep reuse [23, 53]. We first apply our approach to only a single convolutional layer to measure the single-layer speedups (Section 7.2). 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 7.3). 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 7.5). Fourth, we analyze the runtime overhead of each part of our approach (Section 7.6).

Platforms. We conduct experiments to measure the performance of DREW on two experimental platforms. The first platform is equipped with an Intel i7-7700K with 64GB DDR4 memory. The second platform is equipped with an Intel i9-9900K with 64GB DDR4 memory.

Workloads. We evaluate DREW with three different networks: LeNet-5 [25], CifarNet [4], and VGG-16 [36], which are classic and have been evaluated in many works [32, 39, 45, 62]. The dataset of LeNet-5 is MNIST [24] with sparse images of size 28×28. The dataset of CifarNet is CIFAR10 [22] 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-16 is ImageNet [35] with image size of 224×224. VGG-16 uses 3×3 filters exclusively in the convolution layers where the Winograd algorithm can be directly applied.

7.2 Single-Layer Performance

We perform experiments with a range of different clustering configurations and collect the speedup and accuracy results for 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 this and next sections, we fix the batch size to 100; the number of threads on Core i7 platform is set to 8 and on Core i9 platform is set to 16, which are the maximum number of threads of the CPUs.

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 1 the speedups achieved by the configuration that has the highest efficiency score on 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 6.92× on Core i7 platform and 6.45× on Core i9 platform. Even for the highly parallel implementation, DREW still provides an average speedup of 2.14× on Core i7 platform and 1.99× on Core i9 platform. The performance speedup is up to 10.97× and 3.27× 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 more speedups, the accuracy loss could also be large, which results in poor efficiency score (detailed in Section 7.5).

Third, on the networks whose image size is large, such as 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-16, such as Conv3-1, Conv4-1, and Conv5-1, may have a larger remaining ratio compared to the other layers, because of the previous maximum pooling operation.

Accuracy loss. We report the single-layer accuracy loss of DREW compared to [32] in Table 1, which implies that DREW incurs less than 1.32% accuracy loss in each layer.

7.3 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

Table 1: Single-layer performance benefits. *Conf.* **means configuration,** 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,** *i7* **and** *i9* **are** *Core i7* **and** *Core i9* **platforms, and** Δ **Acc is the accuracy loss.**

Network	Layer	Conf.		r_c	Serial		Parallel		Δ Acc
		L_{cb}	$\hat{H}$		i7	i9	i7	i9	
LeNet-5	Conv1	1	16	0.04	4.71×	4.51×	1.16×	1.11×	0.0008
	Conv2	1	12	0.14	5.05×	4.50×	1.42×	1.27×	0.0012
Average					4.88×	4.51×	1.29×	1.19×	0.0010
CifarNet	Conv1	1	16	0.14	6.52×	6.21×	1.77×	1.69×	0.0031
	Conv2	1	12	0.13	5.78×	5.67×	1.66×	1.63×	0.0027
Average					6.15×	5.94×	1.72×	1.66×	0.0029
VGG-16	Conv1-1	1	24	0.05	7.67×	7.21×	2.23×	2.09×	0.0057
	Conv1-2	1	23	0.02	10.97×	9.18×	3.27×	2.74×	0.0075
	Conv2-1	1	22	0.09	8.40×	6.88×	2.76×	2.26×	0.0050
	Conv2-2	1	20	0.07	9.00×	8.29×	2.70×	2.49×	0.0018
	Conv3-1	1	21	0.20	5.08×	4.97×	1.64×	1.61×	0.0037
	Conv3-2	1	18	0.07	8.51×	7.94×	2.74×	2.55×	0.0012
	Conv3-3	1	16	0.06	9.19×	8.57×	2.96×	2.76×	0.0011
	Conv4-1	1	17	0.16	5.92×	5.70×	1.90×	1.83×	0.0019
	Conv4-2	1	15	0.08	6.77×	6.43×	2.22×	2.11×	0.0010
	Conv4-3	1	17	0.09	6.60×	6.35×	2.17×	2.09×	0.0006
	Conv5-1	1	16	0.20	4.71×	4.56×	1.54×	1.49×	0.0002
	Conv5-2	1	16	0.18	4.98×	5.03×	1.63×	1.65X	0.0005
	Conv5-3	1	11	0.07	7.68×	7.53×	2.52×	2.47×	0.0009
Average					7.35×	6.82×	2.33×	2.16×	0.0024

configuration that can attain the best efficiency score mentioned in Section 7.2.

We show our performance benefits on the end-to-end execution time of the full neural networks in Table 2. We involve all runtime overhead (clustering and others) in our time measurement and have the following observations.

Table 2: End-to-end performance benefits. Δ **Acc is the accuracy loss.**

Network	Serial speedup		Parallel speedup		Δ Acc
	Core i7	Core i9	Core i7	Core i9	
LeNet-5	4.39×	4.28×	1.18×	1.15×	0.0026
CifarNet	4.78×	4.34×	1.41×	1.36×	0.0064
VGG-16	9.22×	8.51×	2.76×	2.41×	0.0123

First, our deep-reuse Winograd convolution achieves significant performance benefits. For the serial Winograd convolution, DREW achieves an average speedup of 6.13× on Core i7 platform and 5.71× on Core i9 platform. For the parallel implementation, it still provides an average speedup of 1.78× on Core i7 platform and 1.64× on Core i9 platform. The performance speedup is up to 9.22× and 2.76× in serial mode and parallel mode respectively.

Second, for LeNet-5 and CifarNet, 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. In contrast, for 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 1, 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 1.23%, as shown in Table 2.

7.4 Trade-off between Accuracy and Efficiency

In our experiment, DREW incurs an additional 0.26-1.23% 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 [11]. We re-train the model using DREW. Therefore, only the model weights are updated, not the model. 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-16 cause the major accuracy loss. We can apply deep reuse in the other layers while remaining the front layers unchanged.

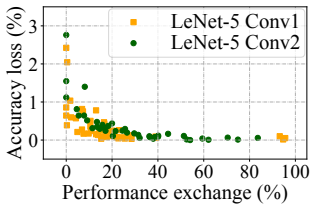


Figure 5: Trade-off between accuracy and performance.

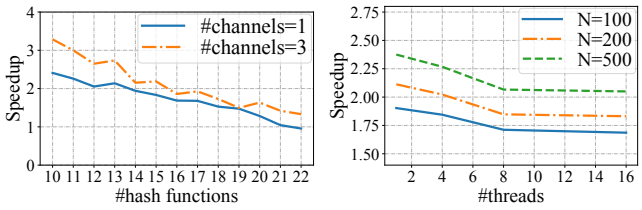
Third, the balance between performance and accuracy loss can be adjusted to meet various application scenarios. The number of channels L_{cb} and the number of hash functions $\hat{H}$, mentioned in Section 6, can be adjusted by users. For example, in a rigid scenario, users could set a small L_{cb} with a large $\hat{H}$ so that the accuracy loss shall be greatly minimized. We use LeNet-5 with 100 batch size on Core i7 platform for illustration, and show the relation between accuracy loss and performance exchange of DREW in Figure 5. The performance exchange is defined as the ratio difference between the performance of each measurement and the highest performance achieved. Figure 5 shows that we can trade performance for accuracy in DREW.

7.5 Configuration Influence Analysis

To further analyze the efficacy of DREW, we explore the influence of different configurations on DREW and use CifarNet *Conv1* 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 change the batch size and the number of threads, while keeping the other configurations unchanged. The performance results are shown in Figure 6.

Hash size. The number of hash functions mainly influences the remaining ratio. With the decreasing number of hash functions, more computation is saved due to a smaller remaining ratio. Therefore, the performance increases along with the hash size.

Clustering granularity. To study how clustering granularity affects the performance, we explore the channel number of tiles L_{cb}



(a) Clustering granularity and the number of hash functions. (b) Batch size and the number of threads.

Figure 6: Influence from different configurations.

of all the factors of C in *Conv1*. Figure 6 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.

Number of threads. DREW achieves significant performance benefits in all cases. In parallel execution, the speedup can be up to 2.38×. 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.

7.6 Execution Time Analysis

We perform runtime analysis for each step of DREW in each layer, and use CifarNet for illustration; the results of the other benchmarks are similar. We divide the workflow of our approach into three stages based on the steps in Section 5.1. The first stage is *clustering*, which represents the step 1 of clustering in Section 5.1. The second stage is *convolution*, which represents the steps 2 to 4. The third stage is *accumulation*, which represents the step 5 of accumulation.

The proportions of clustering, convolution, and accumulation are 16%, 34%, and 50% for *Conv*1, and are 20%, 56%, and 24% for *Conv*2, respectively. We have the following findings. First, the clustering stage occupies the minimum proportion of the overall execution in both layers. Second, for *Conv2*, the convolution stage takes a relatively high proportion because there are more input channels and the number of neuron vectors is larger, resulting in more multiplication operations. Third, the accumulation takes a relatively long time because the memory access to the centroid vector is discontinuous. If users pursue higher performance, it can be optimized by increasing L_{cb}, which reduces the number of discontinuous memory accesses.

7.7 Applicability to Other Networks

DREW can be applied to other convolutional neural networks. We select LeNet-5, CifarNet, and VGG-16 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 [36], which has 64 input features and 64 output features, is the same as Conv2_1 in ResNet-18 [15]. Furthermore, advanced models only make the weights more efficient without considering the input similarities. Therefore, our work still can be used in other situations.

8 RELATED WORK

Winograd. CNNs have been widely applied in Web applications [8, 12, 27, 28, 30, 34, 40, 44, 47, 51]. Cook [7] and Toom [38] proposed a class of minimal filtering algorithms, and Winograd [46, 49] generalized these fast CNN algorithms. Lavin [23] further extended Winograd as an efficient convolution operation. There are many works on accelerating Winograd algorithm. Jia *et al.* [20] optimized the Winograd-based convolution on Intel Xeon Phi platforms. Yan *et al.* [50] optimized the batched Winograd algorithm on GPUs. Moreover, Winograd algorithm has been integrated in many popular libraries, such as FALCON [2], LIBXSMM [5], MKL-DNN [3], and SASS [6].

Data reuse. Data reuse has been proved to be a great success in data management and analytics [33, 54, 56–59], and has been applied in deep learning, so called deep reuse. Deep reuse is an accelerating method that speedups convolution by reusing the similarities among neuron vectors. The closest work to our deep reuse in Winograd is [32]. Ning *et al.* [32] applied the deep reuse techniques in the CNN inference process. Ning *et al.* [31] also applied the deep reuse in CNN training process on the fly. Different from these works, we apply deep reuse to Winograd algorithms, which further improves the performance of CNN. Zheng *et al.* [64, 65] explore data reuse between memory intensive operations for stitching optimization, which is orthogonal to our study.

Compression techniques. Deep reuse, which leverages the similarities among input online, is orthogonal to compression techniques (like pruning and quantization) [9, 14, 17, 18, 21, 42, 48, 52, 61], which reduce the model size offline by leveraging the similarities among weights. Deep reuse can be applied to compressed models, as discussed in [32], and hence they are complementary to each other.

9 CONCLUSION

This paper integrates deep reuse with Winograd convolution, and yields a library, called DREW, to enable efficient CNN inference. By enabling deep reuse in the Winograd algorithm, DREW reduces the number of convolution operations to an average of 11% of the original operations. The paper presents how deep reuse can be applied to Winograd. 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 6.68× performance improvement over the original Winograd convolution with almost no accuracy loss.

ACKNOWLEDGMENTS

This work is supported by the National Key Research and Development Program of China (No. 2018YFB1004401), National Natural Science Foundation of China (No. 62072458, 62172419, and 62072459), and Alibaba Group through Alibaba Innovative Research Program. Feng Zhang is the corresponding author of this paper.

REFERENCES

[1] 2014. cuDNN: Efficient Primitives for Deep Learning. https://developer.nvidia.com/cudnn.

[2] 2016. FALCON Library: Fast Image Convolution in Neural Networks on Intel Architecture. https://colfaxresearch.com/falcon-library/.

[3] 2016. Intel(R) Math Kernel Library for Deep Neural Networks. https://github.com/oneapi-src/oneDNN.

[4] 2020. CifarNet. http://places.csail.mit.edu/deepscene/small-projects/TRN-pytorch-pose/model_zoo/models/slim/nets/cifarnet.py.

[5] 2020. LIBXSMM. https://github.com/hfp/libxsmm.

[6] 2020. MaxAs. https://github.com/NervanaSystems/maxas.

[7] SA Cook. 1966. *On the minimum computation time for multiplication.* Ph.D. Dissertation. Harvard U., Cambridge, Mass.

[8] Rahul Duggal, Scott Freitas, Cao Xiao, Duen Horng Chau, and Jimeng Sun. 2020. REST: Robust and Efficient Neural Networks for Sleep Monitoring in the Wild. In *WWW '20: The Web Conference*, Yennun Huang, Irwin King, Tie-Yan Liu, and Maarten van Steen (Eds.).

[9] Boyuan Feng, Yuke Wang, Tong Geng, Ang Li, and Yufei Ding. 2021. APNN-TC: Accelerating arbitrary precision neural networks on ampere GPU tensor cores. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis.*

[10] Boyuan Feng, Yuke Wang, Gushu Li, Yuan Xie, and Yufei Ding. 2021. Palleon: A Runtime System for Efficient Video Processing toward Dynamic Class Skew. In *USENIX ATC 21.*

[11] Mikhail Figurnov, Aizhan Ibraimova, Dmitry P Vetrov, and Pushmeet Kohli. 2016. PerforatedCNNs: Acceleration through Elimination of Redundant Convolutions. In *Advances in Neural Information Processing Systems 29.*

[12] Junyi Gao, Cao Xiao, Yasha Wang, Wen Tang, Lucas M. Glass, and Jimeng Sun. 2020. StageNet: Stage-Aware Neural Networks for Health Risk Prediction. In *WWW '20: The Web Conference*, Yennun Huang, Irwin King, Tie-Yan Liu, and Maarten van Steen (Eds.).

[13] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. 1999. Similarity Search in High Dimensions via Hashing. In *PVLDB.*

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

[15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *CVPR.*

[16] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *CoRR* abs/1704.04861 (2017). arXiv:1704.04861

[17] Zhenbo Hu, Xiangyu Zou, Wen Xia, Sian Jin, Dingwen Tao, Yang Liu, Weizhe Zhang, and Zheng Zhang. 2020. Delta-DNN: Efficiently Compressing Deep Neural Networks via Exploiting Floats Similarity. In *ICPP.*

[18] Zhenbo Hu, Xiangyu Zou, Wen Xia, Yuhong Zhao, Weizhe Zhang, and Donglei Wu. 2021. Smart-DNN: Efficiently Reducing the Memory Requirements of Running Deep Neural Networks on Resource-constrained Platforms. In *ICCD.*

[19] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. 2018. Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference. In *CVPR.*

[20] Zhen Jia, Aleksandar Zlateski, Fredo Durand, and Kai Li. 2018. Optimizing N-Dimensional, Winograd-Based Convolution for Manycore CPUs. *PPoPP* (2018).

[21] Sian Jin, Chengming Zhang, Xintong Jiang, Yunhe Feng, Hui Guan, Guanpeng Li, Shuaiwen Leon Song, and Dingwen Tao. 2022. COMET: A Novel Memory-Efficient Deep Learning Training Framework by Using Error-Bounded Lossy Compression. *PVLDB* (2022).

[22] Alex Krizhevsky. 2012. Learning Multiple Layers of Features from Tiny Images. *University of Toronto* (05 2012).

[23] Andrew Lavin and Scott Gray. 2016. Fast Algorithms for Convolutional Neural Networks. In *CVPR.*

[24] Yann LeCun, Corinna Cortes, and Christopher J.C. Burges. 1998. THE MNIST DATABASE of handwritten digits. http://yann.lecun.com/exdb/mnist/.

[25] Yann Lecun, L.D. Jackel, Leon Bottou, A. Brunot, Corinna Cortes, J. S. Denker, Harris Drucker, I. Guyon, U.A. Muller, Eduard Sackinger, Patrice Simard, and V. Vapnik. 1995. Comparison of learning algorithms for handwritten digit recognition. In *International Conference on Artificial Neural Networks, Paris*, F. Fogelman and P. Gallinari (Eds.).

[26] Yu-Chiang Li, Chia-Ming Yeh, and Chin-Chen Chang. 2010. Data hiding based on the similarity between neighboring pixels with reversibility. *Digital Signal Processing* (2010).

[27] Bin Liu, Ruiming Tang, Yingzhi Chen, Jinkai Yu, Huifeng Guo, and Yuzhou Zhang. 2019. Feature Generation by Convolutional Neural Network for Click-Through Rate Prediction. In *The World Wide Web Conference*, Ling Liu, Ryen W. White, Amin Mantrach, Fabrizio Silvestri, Julian J. McAuley, Ricardo Baeza-Yates, and Leila Zia (Eds.).

[28] Junxin Liu, Fangzhao Wu, Chuhan Wu, Yongfeng Huang, and Xing Xie. 2019. Neural Chinese Word Segmentation with Lexicon and Unlabeled Data via Posterior Regularization. In *The World Wide Web Conference*, Ling Liu, Ryen W. White, Amin Mantrach, Fabrizio Silvestri, Julian J. McAuley, Ricardo Baeza-Yates, and Leila Zia (Eds.).

[29] Yizhi Liu, Yao Wang, Ruofei Yu, Mu Li, Vin Sharma, and Yida Wang. 2019. Optimizing CNN Model Inference on CPUs. In *USENIX ATC.*

[30] Yilin Liu, Shijia Zhang, and Mahanth Gowda. 2021. NeuroPose: 3D Hand Pose Tracking using EMG Wearables. In *WWW '21: The Web Conference*, Jure Leskovec, Marko Grobelnik, Marc Najork, Jie Tang, and Leila Zia (Eds.).

[31] L. Ning, H. Guan, and X. Shen. 2019. Adaptive Deep Reuse: Accelerating CNN Training on the Fly. In *ICDE.*

[32] Lin Ning and Xipeng Shen. 2019. Deep Reuse: Streamline CNN Inference on the Fly via Coarse-Grained Computation Reuse. In *ICS.*

[33] Zaifeng Pan, Feng Zhang, Yanliang Zhou, Jidong Zhai, Xipeng Shen, Onur Mutlu, and Xiaoyong Du. 2021. Exploring data analytics without decompression on embedded GPU systems. *IEEE Transactions on Parallel and Distributed Systems* (2021).

[34] Jathushan Rajasegaran, Naveen Karunanayake, Ashanie Gunathillake, Suranga Seneviratne, and Guillaume Jourjon. 2019. A Multi-modal Neural Embeddings Approach for Detecting Mobile Counterfeit Apps. In *The World Wide Web Conference*, Ling Liu, Ryen W. White, Amin Mantrach, Fabrizio Silvestri, Julian J. McAuley, Ricardo Baeza-Yates, and Leila Zia (Eds.).

[35] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. 2015. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)* (2015).

[36] Karen Simonyan and Andrew Zisserman. 2014. Very Deep Convolutional Networks for Large-Scale Image Recognition. *arXiv 1409.1556* (09 2014).

[37] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V. Le. 2019. MnasNet: Platform-Aware Neural Architecture Search for Mobile. In *CVPR.*

[38] Andrei L Toom. 1963. The complexity of a scheme of functional elements realizing the multiplication of integers. Soviet Mathematics Doklady.

[39] Hongyi Wang, Scott Sievert, Shengchao Liu, Zachary Charles, Dimitris Papailiopoulos, and Stephen Wright. 2018. ATOMO: Communication-efficient Learning via Atomic Sparsification. In *Advances in Neural Information Processing Systems.*

[40] Tianhao Wang and Florian Kerschbaum. 2021. RIGA: Covert and Robust White-Box Watermarking of Deep Neural Networks. In *WWW '21: The Web Conference*, Jure Leskovec, Marko Grobelnik, Marc Najork, Jie Tang, and Leila Zia (Eds.).

[41] Yuke Wang, Boyuan Feng, and Yufei Ding. 2021. DSXplore: Optimizing Convolutional Neural Networks via Sliding-Channel Convolutions. In *IPDPS.*

[42] Yuke Wang, Boyuan Feng, and Yufei Ding. 2022. QGTC: Accelerating Quantized GNN via GPU Tensor Cores. In *PPoPP.*

[43] Yuke Wang, Boyuan Feng, Xueqiao Peng, and Yufei Ding. 2021. An Efficient Quantitative Approach for Optimizing Convolutional Neural Networks. In *CIKM.*

[44] Yanling Wang, Jing Zhang, Shasha Guo, Hongzhi Yin, Cuiping Li, and Hong Chen. 2021. Decoupling representation learning and classification for GNN-based anomaly detection. In *SIGIR.*

[45] Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. 2016. Learning Structured Sparsity in Deep Neural Networks. In *Advances in Neural Information Processing Systems 29.*

[46] S. Winograd. 1980. Arithmetic complexity of computations.

[47] Fangzhao Wu, Junxin Liu, Chuhan Wu, Yongfeng Huang, and Xing Xie. 2019. Neural Chinese Named Entity Recognition via CNN-LSTM-CRF and Joint Training with Word Segmentation. In *The World Wide Web Conference*, Ling Liu, Ryen W. White, Amin Mantrach, Fabrizio Silvestri, Julian J. McAuley, Ricardo Baeza-Yates, and Leila Zia (Eds.).

[48] Jiaxiang Wu, Cong Leng, Yuhang Wang, Qinghao Hu, and Jian Cheng. 2016. Quantized Convolutional Neural Networks for Mobile Devices. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR).*

[49] Ruofan Wu, Feng Zhang, Zhen Zheng, Xiaoyong Du, and Xipeng Shen. 2021. Exploring deep reuse in Winograd CNN inference. In *PPoPP.*

[50] Da Yan, Wei Wang, and Xiaowen Chu. 2020. Optimizing Batched Winograd Convolution on GPUs. In *PPoPP.*

[51] Quanming Yao, Xiangning Chen, James T. Kwok, Yong Li, and Cho-Jui Hsieh. 2020. Efficient Neural Interaction Function Search for Collaborative Filtering. In *WWW '20: The Web Conference*, Yennun Huang, Irwin King, Tie-Yan Liu, and Maarten van Steen (Eds.).

[52] Chengming Zhang, Geng Yuan, Wei Niu, Jiannan Tian, Sian Jin, Donglin Zhuang, Zhe Jiang, Yanzhi Wang, Bin Ren, Shuaiwen Leon Song, et al. 2021. ClickTrain: efficient and accurate end-to-end deep learning training via fine-grained architecture-preserving pruning. In *ICS.*

[53] Chenyang Zhang, Feng Zhang, Xiaoguang Guo, Bingsheng He, Xiao Zhang, and Xiaoyong Du. 2020. iMLBench: A Machine Learning Benchmark Suite for CPU-GPU Integrated Architectures. https://github.com/ChenyangZhang-cs/iMLBench. *IEEE TPDS* (2020).

[54] Feng Zhang, Zaifeng Pan, Yanliang Zhou, Jidong Zhai, Xipeng Shen, Onur Mutlu, and Xiaoyong Du. 2021. G-TADOC: Enabling Efficient GPU-Based Text Analytics

without Decompression. In *ICDE*.

[55] Feng Zhang, Jidong Zhai, Bingsheng He, Shuhao Zhang, and Wenguang Chen. 2016. Understanding co-running behaviors on integrated CPU/GPU architectures. *IEEE Transactions on Parallel and Distributed Systems* (2016).

[56] Feng Zhang, Jidong Zhai, Xipeng Shen, Onur Mutlu, and Wenguang Chen. 2018. Efficient document analytics on compressed data: Method, challenges, algorithms, insights. *Proceedings of the VLDB Endowment* (2018).

[57] F. Zhang, J. Zhai, X. Shen, O. Mutlu, and X. Du. 2020. Enabling Efficient Random Access to Hierarchically-Compressed Data. In *ICDE*.

[58] Feng Zhang, Jidong Zhai, Xipeng Shen, Onur Mutlu, and Xiaoyong Du. 2022. POCLib: A High-Performance Framework for Enabling Near Orthogonal Processing on Compression. *IEEE Transactions on Parallel and Distributed Systems* (2022).

[59] Feng Zhang, Jidong Zhai, Xipeng Shen, Dalin Wang, Zheng Chen, Onur Mutlu, Wenguang Chen, and Xiaoyong Du. 2020. TADOC: Text analytics directly on compression. *The VLDB Journal* (2020).

[60] Letian Zhang, Lixing Chen, and Jie Xu. 2021. Autodidactic Neurosurgeon: Collaborative Deep Inference for Mobile Edge Intelligence via Online Learning. In *WWW '21: The Web Conference*, Jure Leskovec, Marko Grobelnik, Marc Najork, Jie Tang, and Leila Zia (Eds.).

[61] Shuyu Zhang, Donglei Wu, Haoyu Jin, Xiangyu Zou, Wen Xia, and Xiaojia Huang. 2021. QD-Compressor: a Quantization-based Delta Compression Framework for Deep Neural Networks. In *ICCD*.

[62] X. Zhang, J. Zou, K. He, and J. Sun. 2016. Accelerating Very Deep Convolutional Networks for Classification and Detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2016).

[63] Yipeng Zhang, Bo Du, Lefei Zhang, and Jia Wu. 2020. Parallel DNN Inference Framework Leveraging a Compact RISC-V ISA-Based Multi-Core System. In *KDD*.

[64] Zhen Zheng, Xuanda Yang, Pengzhan Zhao, Guoping Long, Kai Zhu, Feiwen Zhu, Wenyi Zhao, Xiaoyong Liu, Jun Yang, Jidong Zhai, Shuaiwen Leon Song, and Wei Lin. 2022. AStitch: Enabling A New Multi-Dimensional Optimization Space for Memory-Intensive ML Training and Inference on Modern SIMT Architectures. In *Proceedings of the 27th ACM International Conferenceon Architectural Support for Programming Languages and Operating Systems*.

[65] Zhen Zheng, Pengzhan Zhao, Guoping Long, Feiwen Zhu, Kai Zhu, Wenyi Zhao, Lansong Diao, Jun Yang, and Wei Lin. 2020. Fusionstitching: boosting memory intensive computations for deep learning workloads. *arXiv preprint arXiv:2009.10924* (2020).