# Fast Convolution Meets Low Precision: Exploring Efficient Quantized Winograd Convolution on Modern CPUs*†

XUEYING WANG, Beijing University of Posts and Telecommunications, China
GUANGLI LI, Institute of Computing Technology, Chinese Academy of Sciences and University of Chinese Academy of Sciences, China
ZHEN JIA, Amazon Web Services, USA
XIAOBING FENG, Institute of Computing Technology, Chinese Academy of Sciences and University of Chinese Academy of Sciences, China
YIDA WANG, Amazon Web Services, USA

Low-precision computation has emerged as one of the most effective techniques for accelerating convolutional neural networks and has garnered widespread support on modern hardware. Despite its effectiveness in accelerating convolutional neural networks, low-precision computation has not been commonly applied to fast convolutions, such as the Winograd algorithm, due to numerical issues. In this paper, we propose an effective quantized Winograd convolution, named LoWino, which employs an in-side quantization method in the Winograd domain to reduce the precision loss caused by transformations. Meanwhile, we present an efficient implementation that integrates well-designed optimization techniques, allowing us to fully exploit the capabilities of low-precision computation on modern CPUs. We evaluate LoWino on two Intel Xeon Scalable Processor platforms with representative convolutional layers and neural network models. The experimental results demonstrate that our approach can achieve an average of 1.84× and 1.91× operator speedups over state-of-the-art implementations in the vendor library while preserving accuracy loss at a reasonable level.

CCS Concepts: • **Computing methodologies** → **Parallel computing methodologies**; **Neural networks**.

Additional Key Words and Phrases: Deep Learning, Winograd Convolution, Low-Precision Computation.

## 1 INTRODUCTION

Deep convolutional neural networks (CNNs) have achieved remarkable performance in a variety of intelligent tasks, such as object recognition [33] and semantic segmentation [51]. The superior accuracy of CNNs usually comes from complicated model structures [18, 52, 53], which leads to a huge cost of computational complexity. The convolutional layer, which is arguably the most time-consuming component of contemporary CNNs, dominates the runtime performance of model inference. While many efforts have been made in algorithm-level and system-level optimizations [17, 36, 42], the mechanism for effectively optimizing convolution is still a long-standing challenge.

Low-precision computing become a new trend in the deep learning domain, due to the ever-increasing demand for hardware resources and energy, which can accelerate neural networks with negligible precision loss [5]. At the same time, chip vendors have introduced specific instructions

*Extension of Conference Paper. This paper is an extension of the conference paper [13] which was presented at the 50th International Conference on Parallel Processing. This paper extends the previous work by: 1) systematically analyzing the numerical error incurred by existing methods to reveal the challenging problems of low-precision Winograd convolutions; 2) considering both fused and non-fused implementations to further improve the performance; 3) evaluating the proposed low-precision Winograd convolutions on Intel 3rd Xeon Scalable Processors.

†The corresponding author of this paper is Guangli Li (liguangli@ict.ac.cn).

Authors' addresses: Xueying Wang, wangxueying@bupt.edu.cn, Beijing University of Posts and Telecommunications, Beijing, China; Guangli Li, liguangli@ict.ac.cn, Institute of Computing Technology, Chinese Academy of Sciences and University of Chinese Academy of Sciences, 6 Kexueyuan South Rd, Haidian Street, Beijing, China, 100190; Zhen Jia, Amazon Web Services, Santa Clara, CA, USA, zhej@amazon.com; Xiaobing Feng, fxb@ict.ac.cn, Institute of Computing Technology, Chinese Academy of Sciences and University of Chinese Academy of Sciences, Beijing, China; Yida Wang, Amazon Web Services, Santa Clara, CA, USA, wangyida@amazon.com.

for low-precision computing, such as VNNI [23] on x86 architectures and WMMA [46] on Nvidia GPUs, which significantly boost the performance and reduce memory and energy consumption. In order to take full advantage of these instructions, their characteristics should be considered when developing low-precision applications.

On the other hand, fast convolution algorithms, represented by Winograd's minimal filtering algorithm [36], are widely used to accelerate convolutions [28, 43, 58, 59]. The Winograd algorithm reduces the computational complexity by transforming tiles of input images and filters to the Winograd domain and then performing computations there. Theoretically, the reduced computation depends on the tile size. Larger tile sizes result in higher numerical operation reduction, but introduce more numeric errors, due to the numerical instability of the Winograd algorithm [2, 57].

When applying low-precision arithmetic in the Winograd algorithm, the above condition exacerbates and affects the model accuracy, which indicates that fast convolution and quantization are not orthogonal optimization methods that can be combined directly. By investigating existing Winograd convolution implementations [24, 54], we found that only one small tile size, $4 \times 4$, is supported under low-precision computation. The Winograd convolution with small tile sizes can only reduce limited computations in theory and sometimes is slower than direct convolution, since the extra overheads of data movements amortize the benefits of computation savings. A challenging problem that needs to be answered is whether a larger tile size is possible for low-precision Winograd convolutions to improve performance while retaining a reasonable accuracy.

In this paper, we propose LoWino, a novel approach to achieve effective and efficient low-precision Winograd convolutions. We replace the outside-quantization scheme of existing methods [24, 54], which performs quantization in the spatial domain, with an inside-quantization scheme employing a linear quantization process in the Winograd domain, thereby reducing the precision loss incurred by low-precision Winograd transformations. As a result, our methodology solves the problem that only a small tile size is feasible for low-precision Winograd convolution and owns better accuracy than existing solutions. At the implementation level, we employ a customized data layout and design efficient fused and non-fused implementations for low-precision Winograd convolutions. Especially, we combine Winograd computations with quantization operations and employ static scheduling to achieve load-balanced multi-core parallelization, optimizing the performance of the system as a whole. To demonstrate the effectiveness and efficiency, we implement LoWino on Intel platforms with VNNI [23]. Evaluations with representative convolutional layers show that LoWino can achieve remarkable speedups over state-of-the-art implementations. To the best of our knowledge, LoWino has the broadest coverage of Winograd algorithms with efficient implementation in low-precision.

The main contributions of this paper are summarized as follows:

- We analyze the computational errors incurred by combining Winograd convolution with existing quantization schemes and propose a novel quantized Winograd convolution approach, LoWino. Our approach employs an in-side quantization scheme, which performs quantization in the Winograd domain, showing that larger tile size low-precision Winograd convolution can achieve reasonable accuracy.
- We present efficient fused and non-fused implementation of low-precision Winograd convolutions to exploit the low-precision computing capability of modern CPUs, which employs several well-designed optimization techniques to enhance the efficiency in both computation and memory access.
- We evaluate our approach by leveraging representative convolutional layers of prevailing neural networks on Intel Xeon Scalable Processors. The experimental results show that our approach outperforms state-of-the-art methods, which achieves up to 2.90× speedup compared with the best implementations in the vendor library.

## 2 BACKGROUND AND MOTIVATION

### 2.1 Low-Precision Computation

Previous efforts [25, 47, 61] have proved that the memory footprint and computational cost can be reduced through low-precision computation with limited or no accuracy loss. In order to cater to these advantages, hardware vendors have proposed new instruction sets to support low-precision computation. Intel VNNI [23] is a domain-specific instruction set of neural network acceleration, which is an extension of traditional x86 architectures, supporting efficient 8-bit low-precision computations. Theoretically, the 8-bit low-precision computations instruction provides 4× peak performance over FP32 operations. Figure 1 shows the semantic of vpdpbusd, which is a representative 512-bit wide fused-multiply-add (FMA) instruction.

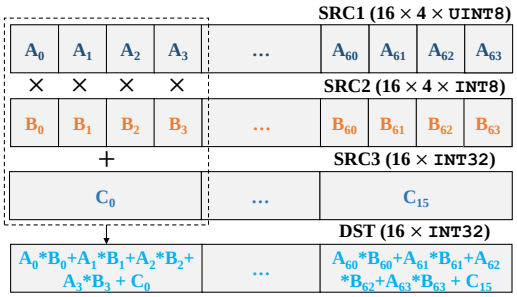



Fig. 1. The semantic of the vpdpbusd instruction.

There are three 512-bit registers $A$, $B$, and $C$ as source operands. $A$ contains 64 unsigned 8-bit integers (UINT8), $B$ contains 64 signed 8-bit integers (INT8), and $C$ contains 16 signed 32-bit integers (INT32). The results are stored in a destination register $D$, which are 16 32-bit integers. In this instruction, to avoid the overflow when accumulating INT8, the element in $C$ and $D$ are INT32 rather than INT8. The vpdpbusd instruction performs the vector dot product between two 4-element vectors, $A_{[i\times4:i\times4+3]}$ and $B_{[i\times4:i\times4+3]}$. The result of the vectors is stored in a 32-bit integer, which will add to the corresponding 32-bit element of $C_i$, and finally accumulated to an element of results. Such a computational pattern of low-precision computation instructions is different from that of general vector instructions. The specific characteristics of low-precision computation must be taken into consideration [56] to take full advantage of low-precision computation instructions.

To represent a full-precision neural network as a low-precision one, quantization [16] usually comes into play, which leverages two operations: a quantization function $Q(x)$ and a de-quantization function $Q'(x)$. The $Q(x)$ quantizes a floating-point value (e.g, FP32) to a low-precision integer value (e.g., INT8), whereas the $Q'(x)$ recovers the low-precision result into the original precision type. A saturated linear quantization function is applied to transform 32-bit floating-point (FP32) to low-precision 8-bit values approximately:

$$Q(X_{\texttt{FP32}}) \quad = \quad (\mathcal{S}_{\texttt{INT8}}(\alpha X_{\texttt{FP32}}))_{\texttt{INT8}} \tag{1}$$

where $\mathcal{S}_{\texttt{INT8}}$ is a saturated transformation function, which represents the FP32 values by integers and limits the values between the fixed upper and lower bounds in Eq. 1. For example, if the value is greater than or less than the upper or lower bound of INT8, it will be set as the maximum or minimum value of the INT8 value range. The original value with full-precision data type (FP32) is quantized to the low-precision data type (INT8) by multiplying the scaling factor $\alpha$, which can be calculated by:

$$\alpha \quad = \quad (2^{b-1}-1)/\tau \tag{2}$$

The bit-width of low-precision data type is represented by $b$, and the range of full-precision values is represented by the threshold value $\tau$, i.e., $(-|\tau| \sim +|\tau|)$. Correspondingly, the de-quantization procedure transforms the low-precision values to full-precision ones and recovers the precision, which can be represented as:

$$Q'(X_{\texttt{INT8}}) \quad = \quad (\alpha^{-1} X_{\texttt{INT8}})_{\texttt{FP32}} \tag{3}$$

Therefore, a low-precision quantized convolution layer can be represented as:

$$\mathcal{Y}_k \quad = \quad \sum_{c=1}^{C} Q'\left(Q\left(\mathcal{G}_{k,c}\right) \circledast Q\left(\mathcal{D}_c\right)\right) \tag{4}$$

where $\mathcal{G}_{k,c}$ denotes the $c$-th channel of the $k$-th filter, $\mathcal{D}_c$ denotes the $c$-th channel of the input tensor, $\mathcal{Y}_k$ denotes the $k$-th channel of the output tensor, and $\circledast$ denotes the low-precision correlation operation. By leveraging the specific low-precision computation instructions, the low-precision operation in Eq. 4 can be computed more efficiently than the full-precision one on modern CPUs.

### 2.2 Winograd Convolution

The theoretical computational complexity for convolution layers can be reduced by Winograd algorithm [36], which can be represented as:

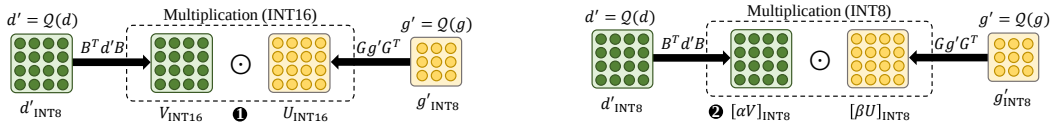
$$\boldsymbol{y}_k \quad = \quad A^{\mathrm{T}}\left(\sum_{c=1}^{C}\left(G\boldsymbol{g}_{k,c}G^{\mathrm{T}}\right) \odot \left(B^{\mathrm{T}}\boldsymbol{d}_c B\right)\right)A \tag{5}$$

where $g$ and $d$ denote the filter and input tile with $C$ input channels and $K$ out channels. The transformation matrices are represented by $A$, $B$, and $G$, and $\odot$ denotes the element-wise multiplication operator between the transformed matrices of the filter and input tile. The $\boldsymbol{g}_{k,c}$ denotes the $c$-th channel of the $k$-th filter; $\boldsymbol{d}_c$ denotes the $c$-th input image tile, and $\boldsymbol{y}_k$ denotes the $k$-th channel of the output result. We follow the convention in [36] and use $F(m \times m, r \times r)$ to represent a 2D Winograd convolution, which takes an input tile size of $(m + r - 1) \times (m + r - 1)$ and filter size of $r \times r$, and generates an output tile of size $m \times m$. The theoretical computational complexity is reduced by $(m + r - 1)^2/(m^2 \times r^2)$. The filter size $r$ is a fixed value in a convolution layer, while the output tile size $m$ is an alterable value. In the computational complexity theory of the Winograd algorithm, the larger the value of $m$, the more computational operations are saved. Nevertheless, the larger $m$ will result in more numeric errors in the final computation result because of the Winograd algorithm's numerical instability [2]. The image sizes in modern convolutional neural networks are beyond the range that the Winograd algorithm can produce sensible results.

The tiling strategy of the Winograd algorithm divides the large input images into small tiles, and each dimension of the tiles exists $r - 1$ overlap. $F(2 \times 2, 3 \times 3)$ and $F(4 \times 4, 3 \times 3)$ are the most widely used combinations of Winograd algorithm parameters for floating-point computations. The Chinese remainder theorem [57] is used to generate Winograd transformation matrices for different tile sizes. For instance, the commonly used input transformation matrices [36] are as follows:

$$B^{\mathrm{T}}_{\langle 2,3\rangle} = \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix}, \qquad B^{\mathrm{T}}_{\langle 4,3\rangle} = \begin{bmatrix} 4 & 0 & -5 & 0 & 1 & 0 \\ 0 & -4 & -4 & 1 & 1 & 0 \\ 0 & 4 & -4 & -1 & 1 & 0 \\ 0 & -2 & -1 & 2 & 1 & 0 \\ 0 & 2 & -1 & -2 & 1 & 0 \\ 0 & 4 & 0 & -5 & 0 & 1 \end{bmatrix}. \tag{6}$$

$B^{\mathrm{T}}_{\langle 2,3\rangle}$ and $B^{\mathrm{T}}_{\langle 4,3\rangle}$ are input transformation matrices for the convolutional filter size $r = 3$, which have tile sizes $m = 2$ and $m = 4$, respectively. After performing $B^{\mathrm{T}}\boldsymbol{d}B$, the values of the input matrix




(a) Up-casting approach. (b) Down-scaling approach.

Fig. 2. State-of-the-art approaches for implementing low-precision Winograd convolution. The green box represents the original/transformed input matrix and the yellow box represents the original/transformed filter matrix. The dashed-line area represents the multiplication operation.

will increase up to 4× and 100× for $F(2\times 2, 3\times 3)$ and $F(4\times 4, 3\times 3)$ on the basis of the coefficients in $B^{\mathrm{T}}$. Compared with the original matrices, the value range of the Winograd-domain inputs will increase, which will further influence the precision. As the tile size increases, the coefficients of the transformation matrices increase dramatically [36], which will cause numerical overflow when performing low-precision Winograd convolutions. Hence, a non-negligible accuracy degradation will incur if applying the low-precision quantization to the Winograd algorithm directly. How to achieve effective implementation of low-precision Winograd convolution is a challenging task.

### 2.3 Existing Approaches

As a running example, we analyze the difference between the state-of-the-art approaches and our low-precision Winograd implementation in Figure 2, depicting the major steps (input transformation, filter transformation, and matrix multiplication) that may result in precision degradation. In this example, we set the problem size to $F(2\times 2, 3\times 3)$, which is one of the most frequently used filter/tile sizes of Winograd convolution in contemporary CNNs. As such, the size of the corresponding input tile $d$ is $4\times 4$ and the size of the filter $g$ is $3\times 3$. In order to apply the low-precision computation to Winograd convolution, a straightforward way is to replace the full-precision matrices with low-precision ones directly. However, as mentioned above, the value range of the matrices will be increased through transformation operations, which may lead to numerical overflow and serious result disturbance. Let us discuss the up-casting approach in ncnn [54] and the down-scaling approach in oneDNN [24], which are two representative approaches for low-precision Winograd convolutions in vendor libraries.

- **Up-Casting Approach.** One potential solution to avoid the overflow caused by transformation operation is up-casting. Figure 2(a) shows an example that up-casts the low-precision data type (e.g., INT8) to a wider data type (e.g., INT16) for transformed matrices (marked as ❶). The higher precision can eliminate the overflow problem, while this multiplication operation may take longer, making it unable to reach the expected acceleration.
- **Down-Scaling Approach.** The down-scaling approach is another solution, it rounds the transformed matrix through a scaling factor multiplication. Since the value range of the input matrix will increase by 4×, according to the input transformation matrix in Eq. 6, we can down-scales the transformed matrix to $round(\alpha V)$ where $\alpha = \frac{1}{4}$, as shown in Figure 2(b). As the down-scaled values suffer from round-off errors, an extra precision loss will be introduced for quantized neural network models (marked as ❷). The scaling factor of input matrices dramatically decreases with the $m$ value used in $F(m\times m, r\times r)$ Winograd algorithm increases, e.g., $\alpha = \frac{1}{4}$, $\frac{1}{100}$, and $\frac{1}{10000}$ for the $m = 2$, $m = 4$, and $m = 6$, respectively (refer to the commonly-used transformation matrices in [36]). The large $m$ brings non-negligible precision loss because of the excessively down-scaling and rounding operations and makes it difficult to apply the down-scaling approach to low-precision Winograd convolutions.

The existing approaches mostly suffer from either performance degradation or precision loss, which motivates our approach for effective implementation of low-precision Winograd convolutions. Despite that there is existing work applying even lower precision integers, such as INT4, to quantize the CNN model [7, 8, 60, 62], the numerical instability of the Winograd algorithm increases the difficulty when applying low-precision computations.

## 3 FROM OUTSIDE-QUANTIZATION TO INSIDE-QUANTIZATION

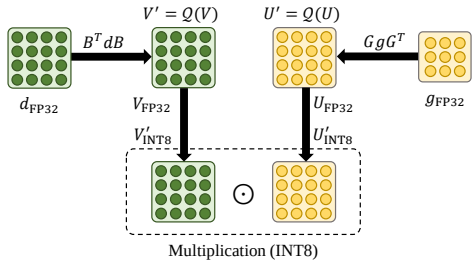



Fig. 3. Inside-quantization approach proposed in this paper.

Existing approaches employ an outside-quantization scheme (i.e., performing quantization outside of the Winograd domain), which inevitably introduces extra precision loss due to the Winograd transformation operations. To alleviate the problem of precision loss, we propose a novel low-precision Winograd algorithm, which employs an inside-quantization scheme instead of the traditional outside-quantization scheme, to efficiently utilize INT8 arithmetical computations on modern processors with low-precision instructions. As shown in Figure 3, the inputs, as well as filters, are retained to the original full-precision data type (FP32), and the transformed matrices are quantized to the low-precision data type (INT8). The quantization is performed in the Winograd domain after the value range is amplified, thereby avoiding numerical overflow in transformation computations. Our low-precision Winograd convolution approach can be represented as

$$\boldsymbol{y}_k = A^{\mathrm{T}}\left(Q'\left(\sum_{c=1}^{C} Q\left(G\boldsymbol{g}_{k,c}G^{\mathrm{T}}\right)\tilde{\odot}Q\left(B^{\mathrm{T}}\boldsymbol{d}_c B\right)\right)\right)A \tag{7}$$

where $Q(x)$ denotes quantization operation, $Q'(x)$ denotes de-quantization operation, and $\tilde{\odot}$ denotes the low-precision element-wise multiplication. As setting the threshold value of $\tau$ in Eq. 2 to $||X_{\mathrm{FP32}}||_\infty$ directly may result in non-neglected accuracy degradation, our approach conducts a calibration process [45] to select a reasonable $\tau$ on tiny amounts of (~500s) unlabeled sample images, which is represented as:

$$\tau = \underset{\tau'}{arg\,min}\left(D_{KL}\left(P\left(X_{\mathrm{FP32}}\right)||P\left(Q_{\tau'}\left(X_{\mathrm{FP32}}\right)\right)\right)\right) \tag{8}$$

$X_{\mathrm{FP32}}$ is the full-precision data and $P(X_{\mathrm{FP32}})$ is the discrete probability distribution of $X$. Correspondingly, $Q(X_{\mathrm{FP32}})$ is quantized data and $P\left(Q_{\tau'}\left(X_{\mathrm{FP32}}\right)\right)$ is the quantized value of $X$. The $D_{KL}$ is the Kullback-Leibler divergence [34] between two probability distributions. The $\tau'$ represents the threshold used in the quantization function $Q(x)$ (refer to Eq. 2). The input of a convolutional layer is collected by executing the neural network on the sample images, while the filters can be directly extracted from the pre-trained neural network model.

## 4 SYSTEM DESIGN

In this section, we describe the implementation design of low-precision Winograd convolution in detail. To achieve high-efficiency implementation for the low-precision quantization Winograd convolution, several challenges need to be considered as follows: *1)* extra memory overhead caused by the non-consecutive memory access from scattering and gathering operations; *2)* overheads from quantization and de-quantization process; *3)* input/output data types and layout requirements from low-precision computation instructions; *4)* sub-optimal performance of irregular-shaped (i.e., tall and skinny) matrix multiplication operations, which has been revealed in prior studies[37].

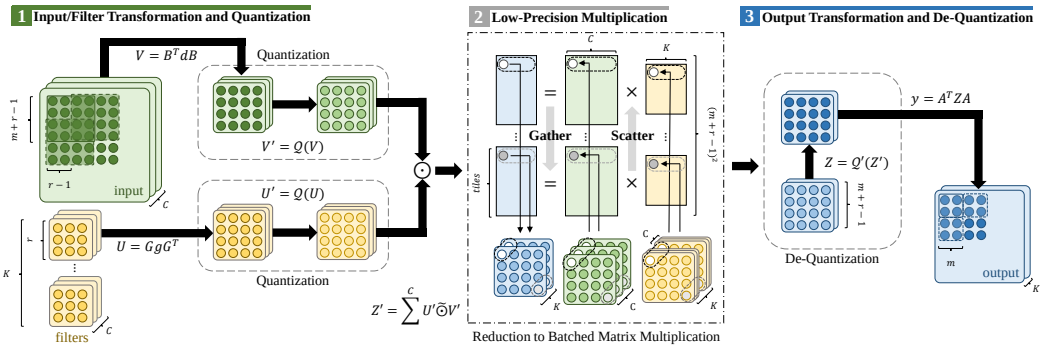



Fig. 4. Overview of LoWino using $F(2 \times 2, 3 \times 3)$ as an example. Green, yellow, and blue boxes represent input tiles, filters, and output tiles, respectively. The dashed rounded-rectangle boxes represent the processes of quantization and de-quantization.

To tackle these challenges, a variety of optimization techniques are applied in different parts of the entire implementation workflow of our low-precision convolution algorithm. As shown in Figure 4, LoWino consists of three main computational phases, including input and filter transformation, low-precision matrix multiplication, and output transformation. We classify the operations into two categories, including memory-bound operations (input, filter, and output transformation) and computation-bound operations (matrix multiplication). To increase the computation efficiency, we guide performance optimizations with the following principles: *a)* Overlapping computation and memory operations as far as possible. *b)* Reducing the latency of memory access. *c)* Increasing the computation efficiency by using data reuse and vectorization. In addition, we implement LoWino in two manners (i.e., non-fused and fused implementations) to further improve the performance, as will be described in Section 4.4.

### 4.1 Data Layout

The data layout stored in memory critically determines the performance. Inspired by the state-of-the-art full-precision approaches [28, 63], we customize a data layout to maximize data reuse and cater to the requirements of low-precision computational instructions. We design the data layout according to the following guidelines: *a)* It should support the low-precision computation instructions, which is the key difference from full-precision design (refer to Section 2.1); *b)* It should support the aligned vector loads and stores, which is essential to fully vectorized programs; *c)* It should be cache-friendly, which is significant in reducing cache and TLB misses by restricting the memory access range.

The customized data layout is illustrated in Table 1. $B$, $C$, $K$, $H$, $W$, $H'$, and $W'$ denote the batch size, input channels, output channels, input image height, input image width, output image height,

Table 1. Customized Data Layout.

| Variable | Data Layout |
|---|---|
| Inputs | $B \times [C/\varphi/\sigma] \times H \times W \times \varphi \times \sigma$ |
| Transformed Inputs | $[N/N_{blk}] \times [C/C_{blk}] \times T \times N_{blk} \times C_{blk}$ |
| Filters | $C \times [K/\varphi/\sigma] \times r \times r \times \varphi \times \sigma$ |
| Transformed Filters | $[C/C_{blk}] \times [K/K_{blk}] \times T \times [C_{blk}/\varphi] \times [K_{blk} \times \varphi]$ |
| Transformed Outputs | $B \times [K/\varphi/\sigma] \times N \times T \times \varphi \times \sigma$ |
| Outputs | $B \times [K/\varphi/\sigma] \times H' \times W' \times \varphi \times \sigma$ |

and output image width, respectively. The filter size is $r \times r$. The $\sigma$ denotes the vector length of the 32-bit floating-point elements and $\varphi$ denotes the number of 8-bit elements in a 32-bit word. For VNNI instruction set, $\sigma = 16$ and $\varphi = 4$. The input image will be divided into $N$ tiles consisting of overlapping elements, and each single input tile has $T$ elements. There are $T$ small matrix multiplications to be conducted in the Winograd algorithm. The blocking hyper-parameters, $C_{blk}$, $K_{blk}$, and $N_{blk}$, are used in matrix multiplication, which will be explained in detail in Section 4.3. Combining our customized data layout with the computational pattern of low-precision instructions allows us to effectively execute matrix multiplication with VNNI instructions. By decoupling the original computation mode into adjacent computation mode, a relatively small range of memory access operations reduces the TLB and cache misses. In addition, to take further advantage of aligned vectorized load/store instructions, all data is 64-byte aligned.

### 4.2 Transformation

Winograd convolution consists of three kinds of transformations. The input transformation and filter transformation (the first phase in Figure 4) transform the original matrix data into the Winograd domain, which is conducted before the matrix multiplications step. The output transformation (the third phase in Figure 4) transforms the result of matrix multiplication and brings it back to the spatial domain. These three transformations are based on a tile of data and the corresponding transformation matrix ($B$, $G$, and $A$). These transformation operations are memory-bound, and the execution time is determined by the data movement mode. To overlap computation and memory operations, we combine compensation, quantization, and de-quantization with transformations, which only introduces negligible overhead and reduces the burden of matrix multiplication. The following subsections describe the implementations and optimization in each transformation stage.

*4.2.1 Input transformation.* Before the matrix multiplication procedure, we should prepare the input data. As described in Section 2.1, we need to quantize 32-bit values to 8-bit integers due to the core instruction in matrix multiplication, `vpdpbusd`, requiring the first operand (i.e., transformed input) to be unsigned type. Originally, each tile $d$ contains 32-bit full-precision values and the data is in the spatial domain. We first transform the data to the Winograd domain and then quantize the floating-point values through function $Q(x)$ in Eq. 1. However, there is no guarantee that the transformed input data is non-negative and can not guarantee that UINT8 can represent all values. Therefore, we add a compensation operation to add 128 to the transformed input data after quantization. In order to eliminate the side-effect introduced by adding 128, we perform subtraction after the matrix multiplication stage to recover the result.

Finally, the result tiles are stored in memory for matrix multiplication (the second phase in Figure 4) and scattered to $T = (m + r - 1)^2$ matrices, leading to non-consecutive memory writes. Here, we utilize several optimizations to overcome the drawbacks of the transformation operations. We store 512-bit data (a whole cache line) in memory each time to eliminate false sharing based on the customized data layout. In addition, we apply non-temporal stores and write data in memory

directly without fetching data to the cache first, in the non-fused implementation. Because the transformed data will be consumed after executing a few instructions, we don't have to worry about breaking cache locations with non-temporary storage.

*4.2.2 Filter Transformation.* The filter transformation operations are performed offline and will not be counted into the execution time due to the transformation being finished ahead of the inference phase on the pre-trained filters. The compensation operation multiplies the result by an auxiliary matrix filled by -128 after quantization to guarantee correctness. To be compatible with the vpdpbusd instruction, the data layout of the filter transformation results will be reorganized, where the shapes of the two lowest dimensions are $(C_{blk}/\varphi) \times (K_{blk} \times \varphi)$.

*4.2.3 Output Transformation.* Winograd algorithm [36] fetches data from $T = (m+r-1)^2$ matrices generated by the matrix multiplication stage, which is called gathering operations, and then performs transformations. However, when gathering operations are triggered, data access is not consecutive and the computation needs to wait until the data is ready in the cache memory. To make all data access consecutive, we alter the gathering operations to the scattering operations at the end of the previous matrix multiplication phase, which will be further explained in Section 4.3. A consecutive 32-bit integer data tile will be de-quantized after the matrix multiplication, which is represented by the function $Q'(x)$. After casting 32-bit integers to floating-point values, we multiply the data tile with the reciprocal of the scaling factor to de-quantize the value and then do the transformation operation by multiplying the corresponding transformation matrix.

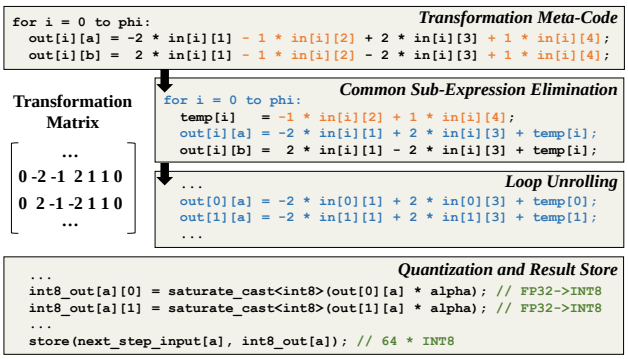



Fig. 5. An example of codelets generation.

*4.2.4 Codelets.* We formulate a codelet generator for Winograd transformations that automatically generates C++ codes for a given $F(m, r)$ based on the transformation matrices provided by wincnn [35]. Figure 5 illustrates an example of codelet generation for input transformation, which operates $(m + r - 1) \times \varphi \times \sigma$ elements in a column-wise or row-wise computation. The codelet for the filter and output transformations is generated and optimized in a similar way. We eliminate the redundant multiplication operation with a zero multiplier caused by many zeros in the transformation matrices. To reduce the computational complexity in the transformation matrix, we compute and store the common terms among different rows into intermediate variables, and replace the common terms in the following expressions with the intermediate variables. To improve the instruction-level parallelism, we unroll the loop of `phi` ($\varphi$) to reduce the branches of instructions in codelets. We perform a column-wise manner and then alternate a row-wise manner on input tiles by utilizing generated codelets, among all the input transformations. Besides, we invoke Intel Intrinsics [22] to keep all the code vectorized.

### 4.3 Batched Matrix Multiplication

The computational efficiency of Winograd convolution is dominated by low-precision matrix multiplication (the second phase in Figure 4). The quantized transformed filter $U'$ and the quantized transformed input tile $V'$ execute low-precision element-wise multiplication in the same channel and accumulate the results along channels (Eq. 7), thus can be reduced to batched matrix multiplication. We perform matrix multiplications $Z = V \times U$ with a batch size of $T = (m+r-1)^2$. The input image transformation with the size of $N \times C$ is represented by matrix $V$ and the filter transformation with the size of $C \times K$ is represented by matrix $U$. A large number of input tiles ($N$) and relatively small input channels ($C$) and output channels ($K$) lead the shape of matrices in matrix multiplication to be tall and skinny. In general, these tall and skinny matrix multiplications lack high optimization in off-the-shelf libraries [28, 55]. We implement batched matrix multiplications that enhance data reuse for both cache memory and registers by blocking strategies. To cooperate with other parts of Winograd convolutions, we perform several specific optimizations for matrix multiplication: *a)* We generate the codes of matrix multiplication according to our customized data layout, so as to fully utilize low-precision instructions; *b)* The results of multiplication are scattered directly into the memory used in the output transformation stage; *c)* For fused and non-fused manners, we adopt different task scheduling strategies to guarantee the order of computations. The optimization of batched matrix multiplications will be described in detail in the following subsections.

*4.3.1 Cache Blocking.* We split the matrix into sub-matrices, and each sub-matrix can be fit in the L2 cache to improve the reusability of data. Figure 6 describes our cache-blocking method. Its main idea is to make full use of data as much as possible before swapping it out of the cache. The matrix $V$ is split into $\lceil N/N_{blk} \rceil \times \lceil C/C_{blk} \rceil$ small sub-matrices of size $N_{blk} \times C_{blk}$, and the matrix $U$ is split into $\lceil C/C_{blk} \rceil \times \lceil K/K_{blk} \rceil$ small sub-matrices of size $C_{blk} \times K_{blk}$. In consequence, the result matrix $Z$ consists of $\lceil N/N_{blk} \rceil \times \lceil K/K_{blk} \rceil$ small sub-matrices of size $N_{blk} \times K_{blk}$. Each sub-matrix in $Z$ can be accumulated by a corresponding row block in $V$ and a corresponding column block in $U$:

$$z_{i,j} = \sum_{k=1}^{C/C_{blk}} v_{i,k} \times u_{k,j} \tag{9}$$

There are $C/C_{blk}$ sub-matrices results of the multiplication operations to be accumulated to the sub-matrix $z$. The intermediate accumulation results are buffered in a $C_{blk} \times K_{blk}$ L2 cache during the multiplication process in Eq. 9. In most cases, the matrix $u$ of size $C_{blk} \times K_{blk}$ can stay in the L2 cache after blocking until all the computations are finished. To improve cache memory utilization, there are two strategies for processing a result sub-matrix $z$: 1) *fused implementation*: performing output transformation for $z$ in the cache directly; 2) *non-fused implementation*: prefetching instructions load $v_{i+1,k}$ into the L2 cache when computing $v_{i,k}$, which can prepare the data in cache before the next matrix multiplication. The details of these two implementations will be described in Section 4.4.

*4.3.2 Register Blocking.* A register-level blocking strategy is designed to optimize the sub-matrices multiplication in Figure 7, which divides original matrices into smaller sub-matrices. The key idea of register-level blocking is similar to cache-level blocking, which will store the intermediate results in $row_{blk} \times col_{blk}$ registers and reuse them. The final results are scattered into the appropriate locations of tiles. As the memory access is consecutive, the hardware prefetcher can automatically prefetch each row of $v$ into the L1 cache. Since the data is stored in a column-major format, each column of $u$ can be prefetched and benefit from the hardware prefetcher. To follow the convention of low-precision computational instruction, i.e., `vpdpbusd`, a specific layout is used to store sub-matrix $u$, which will be reordered to the size of $(C_{blk}/4) \times (K_{blk} \times 4)$.

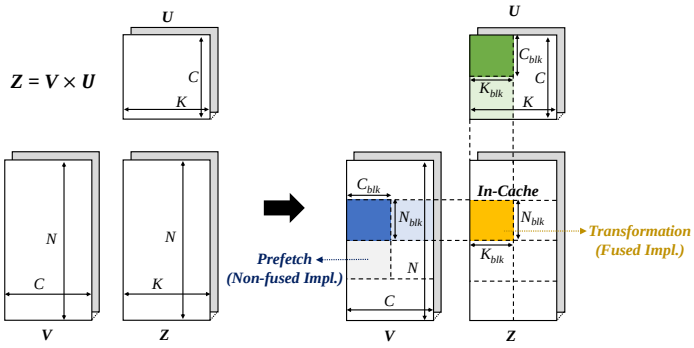



Fig. 6. Cache-level matrix blocking strategy. The blocks in blue, green, and yellow represent the blocked sub-matrix of $V$, $U$ and $Z$, respectively.

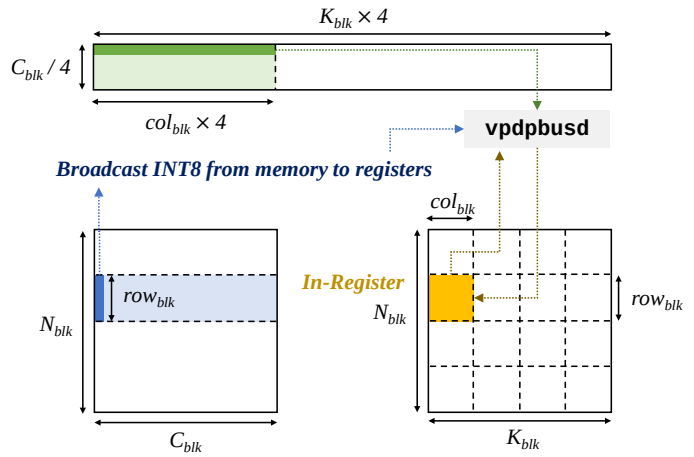



Fig. 7. Register-level blocking. The blocks in blue, green, and yellow represent the blocked sub-matrix of $v$, $u$, and $z$, respectively.

*4.3.3 Compensation.* In the input transformation phase and the filter transformation phase, we conduct compensation operations, including adding 128 and multiplying −128 respectively, for the vpdpbusd instruction requiring the first operand to be unsigned 8-bit integers. The concrete computations we conducted in this phase are depicted in the following equation:

$$Z = \hat{V} \times U + \hat{Z} \quad \text{where} \quad \begin{cases} \hat{V} = V + \Delta \\ \hat{Z} = -\Delta \times U \end{cases} \tag{10}$$

These compensation operations are performed in transformation phases as described in Sections 4.2.1 and 4.2.2. In the input transformation phase, the $\hat{V}$ is calculated by adding an auxiliary matrix $\Delta$, which has the same size as $V$. Correspondingly, the $\hat{Z}$ is calculated by multiplying $-\Delta$ and $U$ in the filter transformation phase. Since the transformation phases are mostly memory-bound, the extra computation has little effect on the performance.

*4.3.4 Code Generation and Tuning.* Due to the configuration of the convolution layer is known at compile time, we can leverage the optimization opportunities about the constants, such as the loops and the memory access offsets. The code for matrix multiplication, which computes $z = v \times u + z$,

is generated by JIT (just-in-time) compilation technique with a specific code template, and the pseudo-code is described in Figure 8.

```
for r0 = 0 to N_blk/row_blk:
  for c0 = 0 to K_blk/col_blk:
    for t = 0 to C_blk:                                  // unroll(t)
      for r1 = 0 to row_blk:
        v_reg = broadcast(v[r0*row_blk +r1][t]);         // unroll(r1)
        prefetch(next_v[r0*row_blk+r1][t],
                 NON_FUSED_FLAG);
        for c1 = 0 to col_blk                            // unroll(c1)
          u_reg[c1] = u[t][c0*col_blk+c1];
          z_reg[r1][c1] = vpdpbusd(z_reg[r1][c1],
                          v_reg, u_reg[c1],);
    for r1 = 0 to row_blk:                               // unroll(r1)
      for c1 = 0 to col_blk:                             // unroll(c1)
        store(output[r0*row_blk+r1][c0*col_blk+c1],
              z_reg[r1][c1]);
```

Fig. 8. The pseudo-code for matrix multiplication.

The loops with loop variables `r0` and `c0` are cache-level blocking and the loops `r1` and `c1` are register-level blocking. A packed 32-bit word (containing four 8-bit integers) is broadcast from the source location of $v$ to a 512-bit vector register `v_reg` in each loop `r1`. In each innermost loop `c1`, we load the operand $u$ into register `u_reg`, which contains 64 8-bit integers, and store the results into register `z_reg`. Significantly, the data of the register `v_reg` in loop `r1` is reused in loop `c1`. To prepare the data for the next computation at the cache line granularity, the results will be scattered to appropriate locations at the end of this phase, which constitutes consecutive memory access and prevents expensive gathering operations from happening. We unroll the loops completely, including loop `t`, `r1`, and `c1`, with the purpose of making instructions consecutive and improving the performance further. To improve the instruction parallelism, we reorder the unrolled instructions to mix the computation instructions and load/store instructions. In addition, for non-fused implementation, we insert a prefetch instruction so as to decrease the memory accessing latency for the next $v$. The code is generated for a specific matrix multiplication operation, which will be compiled into a shared library for all sub-matrices computations.

The parameters $N_{blk}$, $C_{blk}$, $K_{blk}$, $row_{blk}$, and $col_{blk}$ are designed for tuning the generated code. Due to the known configuration of the convolution layers, the auto-tuning approach is conducted ahead of time to find the optimal parameters, which only takes a slight overhead and achieves relatively high performance. The optimal parameters are saved into a wisdom file and used in inference. We limit $row_{blk} \times col_{blk} + col_{blk}$ to less than 31 to reduce the search space. Although there are 32 512-bit vector registers available on modern x86 platforms, we still need to set the limitation to 31 because an extra auxiliary register needs to be reserved for broadcast operations. Furthermore, $C_{blk} \times K_{blk}$ is limited to less than $512^2$ to ensure that the cache memory can hold the sub-matrices, including $z$, $u$, and $v$.

### 4.4 Fused and Non-Fused Implementations

Winograd convolutions can be implemented in two different ways, non-fused implementations and fused implementations, as shown in Figure 9 and Figure 10. While the non-fused implementation is preferred on CPU platforms in prior studies [28, 29] and shows good performance, we find that the fused implementation performs better than the non-fused one in some cases with small tile sizes. Therefore, we implemented two versions of LoWino, which employ non-fused and fused manners, respectively, including their individual transformation codelets and multiplication code

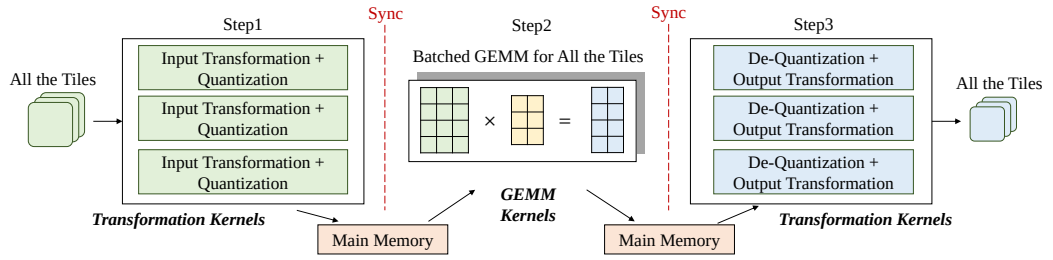



Fig. 9. Non-fused implementation for Winograd convolution.

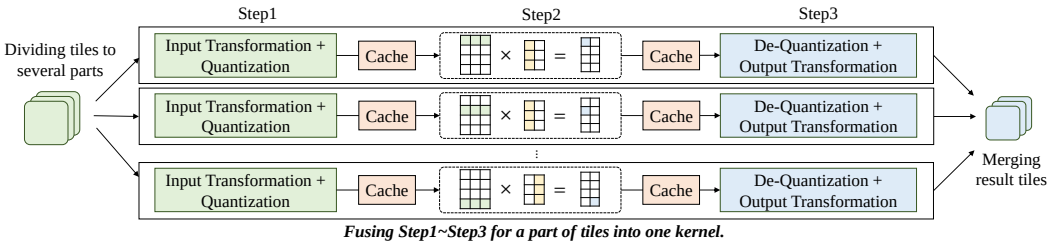



Fig. 10. Fused implementation for Winograd convolution.

templates. These two variants (non-fused and fused) of implementations for low-precision quantized Winograd convolution may perform differently on diverse hardware platforms. We select the best implementation for each convolution layer by utilizing a search process on a specific target platform, so as to achieve optimal model performance.

**Non-Fused Implementations.** In the non-fused implementation of LoWino, each stage uses a separate kernel that fetches all the input tiles and writes all the results to the main memory, as shown in Figure 9. We treat each stage of the Winograd algorithm as an individual sub-task and insert a synchronous operation at the end of the sub-task, thereby guaranteeing its correctness. While the non-fused version introduces the inevitable overhead of data movement between the cache and main memory, it provides good performance when using large tiles for Winograd convolution. By considering the shape and layout of all the input tiles simultaneously, the most time-consuming computation (i.e. matrix multiplication) can be optimized, and computational resources can be better utilized.

**Fused Implementations.** In the fused implementation of LoWino, the input transformation stage, the multiplication stage, and the output transformation stage for a part of input tiles are fused into a single kernel, as shown in Figure 10. In our design, the number of tiles to be processed in a fused kernel is controlled by the blocking hyper-parameters $N_{blk}$ and $K_{blk}$ in the matrix multiplication stage, and the number of tiles to be processed of input and output of transformation stages can be inferred accordingly. The fused version of low-precision Winograd convolution can potentially improve the utilization of the cache, avoiding the movement of intermediate data. However, due to the limitation of the cache memory capacity of hardware platforms, the fused implementations exhibit higher performance at small tile sizes such as $F(2, 3)$.

### 4.5 Parallelization on Multi-Core Processors

To parallel the computation of the low-precision Winograd convolution algorithm, we need to distribute the computation tasks to multi-core processors ahead of execution. Considering the

known and fixed configurations of convolutional layers, we design a static scheduling strategy to deliver the jobs to each thread at compile-time to reduce run-time overheads. Under the same memory access patterns, the amount of computation assigned to each thread is roughly similar, which helps to pursue optimal performance.

In our implementation, we assume that there are $\omega$ threads executed on multi-core platforms in parallel. For fused implementation, $\lceil N/N_{blk} \rceil \times \lceil K/K_{blk} \rceil$ input parts are divided into $\omega$ parts equably, and each thread performs a fused kernel of all the stages for up to $\lceil (\lceil N/N_{blk} \rceil \times \lceil K/K_{blk} \rceil)/\omega \rceil$ parts of tiles. For non-fused implementation, the parallelization method is described as follows. During the input and output transformation phases, there are $N$ tiles that need to be processed simultaneously in total. Each tile contains $T \times \varphi \times \sigma$ elements and each thread needs to compute $\lceil N/\omega \rceil$ tile for transformation tasks. To balance the computation of each thread, the task dimensions will be recursively divided and make more chances for cache reuses. In both the input transformation stage and filter transformation stage, each thread operates $\lceil (C \times K/\varphi/\sigma)/\omega \rceil$ tasks. Our blocking strategies distribute $N_{blk} \times K_{blk}$ elements to each sub-matrix, which is illustrated in Section 4.3. Totally, there are $N/N_{blk} \times K/K_{blk} \times T$ sub-matrices multiplications in the matrix multiplication stage. There are $\lceil (N/N_{blk} \times K/K_{blk} \times T)/\omega \rceil$ matrix multiplication jobs that need to be done in each thread.

The parallel jobs are managed by a single fork-join method. Typically, parallel execution performance depends on the time it takes from the first job start to the last job finish. If all threads start and finish at the same time, it can lead to a balanced situation. Due to the number of $C$, $K$, and $\omega$ being powers of two in general, the workloads can be well balanced among all the threads.

## 5 EVALUATION

In this section, we demonstrate that LoWino is an effective approach for accelerating Winograd convolutions by utilizing low-precision computations while maintaining accuracy at a reasonable level. Specifically, we address two major Research Questions (RQ):

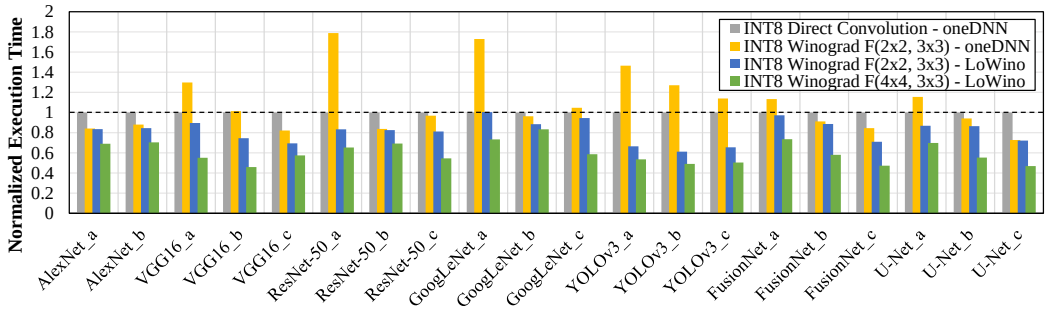
**RQ1.** What is the performance of LoWino compared with the state-of-the-art implementations?

**RQ2.** What is the accuracy loss of our approach for representative convolutional neural networks?

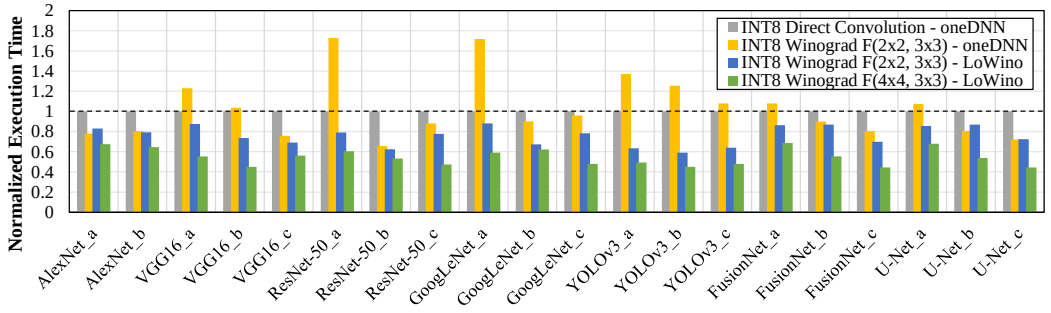
**Experimental Setup.** We perform these experiments on two CPU platforms, an 8-core Intel 2nd Xeon Scalable Processor platform, and an 8-core Intel 3rd Xeon Scalable Processor platform. The platforms both run Linux-based operating systems, Ubuntu20.04 LTS, and have 32GB global main memory. These programs are compiled by g++ (version 9.3.0). We leverage the oneDNN library, a state-of-the-art vendor library on Intel CPU platforms, as our baseline of INT8 Winograd convolutions. The oneDNN library is well-tuned and optimized for Intel CPU architectures and supports the latest hardware features such as low-precision computing. Moreover, oneDNN can effectively support INT8 Winograd convolutions which employs a down-scaling approach. To reduce the interference of initialization, we warm up the experiments and run tests 100 times, and report the average running time.

### 5.1 Convolutional Layer Speedups

We evaluate our implementations on representative convolutional layers in prevailing neural network models including image classification, object detection, and semantic segmentation. To be specific, our benchmarks cover AlexNet [33], VGG16 [52], ResNet-50 [18], GoogLeNet [53], YOLOv3 [50], FusionNet [49], and U-Net [51]. Following the convention [28], we set the batch size for classification models to 64 and the batch size for object detection and semantic segmentation models to one. For each neural network, we collect two or three configurations of layers, which are frequently used configurations in the model. The different layer in a model is named with the suffix




(a) Performance compassion on the 2nd Xeon Scalable Processor platform.




(b) Performance compassion on the 3rd Xeon Scalable Processor platform.

Fig. 11. Normalized execution time for different convolution layers.

a/b/c. The specification of these convolutional layers is described in Table 2. We compare LoWino with the state-of-the-art low-precision convolution implementations in the oneDNN library [24], including direct and Winograd convolution, which include the quantize and de-quantize steps.

Table 2. Benchmarked 3×3 Convolutional Layers.

| Image Classification | | | | | Object Detection | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **Layer** | **Batch** | **C** | **K** | **H&W** | **Layer** | **Batch** | **C** | **K** | **H&W** |
| AlexNet_a | 64 | 384 | 384 | 13 | YOLOv3_a | 1 | 64 | 128 | 64 |
| AlexNet_b | 64 | 384 | 256 | 13 | YOLOv4_b | 1 | 128 | 256 | 32 |
| VGG_a | 64 | 256 | 256 | 58 | YOLOv5_c | 1 | 256 | 512 | 16 |
| VGG_b | 64 | 512 | 512 | 30 | **Semantic Segmentation** | | | | |
| VGG_c | 64 | 512 | 512 | 16 | **Layer** | **Batch** | **C** | **K** | **H&W** |
| ResNet_a | 64 | 128 | 128 | 28 | FusionNet_a | 1 | 128 | 128 | 320 |
| ResNet_b | 64 | 256 | 256 | 14 | FusionNet_b | 1 | 256 | 256 | 160 |
| ResNet_c | 64 | 512 | 512 | 7 | FusionNet_c | 1 | 512 | 512 | 80 |
| GoogLeNet_a | 64 | 128 | 192 | 28 | U-Net_a | 1 | 128 | 128 | 282 |
| GoogLeNet_b | 64 | 128 | 256 | 14 | U-Net_b | 1 | 256 | 256 | 138 |
| GoogLeNet_c | 64 | 192 | 384 | 7 | U-Net_c | 1 | 512 | 512 | 66 |

Figure 11 show the normalized execution time for all the convolutional layers in Table 2, on two evaluation platforms. Overall, LoWino $F(4 \times 4, 3 \times 3)$ achieves up to 2.74× and 2.90× speedups and an average of 1.84× and 1.91× speedups over the best implementations in oneDNN, on the two platform, respectively. There are three observations. First, LoWino $F(2 \times 2, 3 \times 3)$ achieves

Table 3. The end-to-end top-1 accuracy of CNNs with our approach on the ImageNet dataset.

| Model | Method | | FP32 Acc. (%) | INT8 Acc. (%) |
|---|---|---|---|---|
| VGG16 [52] | Non-Winograd Covolution | KLD [45] | 69.40 | 69.20 |
| | | Yao *et al.* [61] | 69.40 | 69.10 |
| | $F(2\times2, 3\times3)$ | oneDNN [24] | 71.59 | 70.98 |
| | | LoWino (Ours) | 71.59 | 71.33 |
| | $F(4\times4, 3\times3)$ | Down-Scaling Impl. | 71.59 | 00.00 |
| | | LoWino (Ours) | 71.59 | 69.20 |
| ResNet-50 [18] | Non-Winograd Convolution | KLD [45] | 73.23 | 73.10 |
| | | Yao *et al.* [61] | 75.30 | 74.80 |
| | | Jacob *et al.* [25] | 76.40 | 74.90 |
| | | Park *et al.* [47] | 77.72 | 75.67 |
| | | Krishnamoorthi [32] | 75.20 | 75.10 |
| | $F(2\times2, 3\times3)$ | oneDNN [24] | 76.13 | 75.91 |
| | | LoWino (Ours) | 76.13 | 76.09 |
| | $F(4\times4, 3\times3)$ | Down-Scaling Impl. | 76.13 | 00.00 |
| | | LoWino (Ours) | 76.13 | 75.53 |

competitive performance compared with the implementations in the vendor library, i.e., oneDNN's 8-bit Winograd convolution. Second, LoWino $F(4\times4, 3\times3)$ usually is the best performer, delivering significant performance improvement over $F(2\times2, 3\times3)$. The speedups of LoWino are derived from both the theoretical complexity reduction of the Winograd algorithm and our optimization techniques. Third, Winograd convolution does not always outperform direct convolution, despite that the former has lower computational complexity. This is because the memory overhead of transformation operations is non-negligible for some layers, which increases overall execution time. Nevertheless, LoWino accelerates the convolution layers in most cases.

### 5.2 End-to-End Performance of Neural Networks

To evaluate the end-to-end model performance, including accuracy and speedup, we choose VGG16 [52] (single-branch structure) and ResNet-50 [18] (multi-branch structure), two representative convolutional neural networks, as benchmarks. VGG and ResNet neural network structures are arguably two of the most popular backbone structures in various intelligent applications [31]. Besides, as discussed in prior studies [19, 20], the multiple-branch ResNet is less redundant than single-branch networks, which is more difficult to compress and accelerate.

We evaluate the model accuracy of VGG16 and ResNet-50 on the ImageNet dataset [10]. In addition to comparing LoWino with the implementations of the Intel oneDNN library, we also compare our approach with state-of-the-art post-training quantization methods [25, 32, 45, 47, 61] which leverage normal (non-Winograd) low-precision convolutions. As $F(4\times4, 3\times3)$ is not supported in oneDNN, we implement the down-scaling approach ourselves and evaluate its accuracy. Table 3 reports the accuracy of models with different implementations of low-precision convolutions. We use the full-precision model in the official model repository of PyTorch [48] as the baseline. Since the different hyper-parameters used in those papers, e.g., learning rate and data augmentation, the accuracy numbers slightly vary. For fairness, we report not only the accuracy of low-precision models but also the full-precision baseline models. For non-Winograd post-training quantization methods [25, 32, 45, 47, 61], the accuracy numbers are directly cited from the corresponding papers.

The accuracy of quantized neural networks that utilize low-precision Winograd convolutions is similar to those non-Winograd convolutions. Benefiting from our quantization design, LoWino achieves less accuracy loss compared with the down-scaling approach when using $F(2\times2, 3\times3)$. As discussed in Section 2.3, the scaling factor of $F(4\times4, 3\times3)$ is much less than the factor of

Table 4. The end-to-end inference speedups of CNNs with INT8 Winograd convolutions.

| Model | $F(2\times2,3\times3)$ - **oneDNN** | $F(2\times2,3\times3)$ - **LoWino** | $F(4\times4,3\times3)$ - **LoWino** |
|---|---|---|---|
| VGG16 [52] | 1.00 × | 1.34 × | 2.04 × |
| ResNet-50 [18] | 1.00 × | 1.05 × | 1.11 × |

$F(2\times2,3\times3)$, leading to much more precision loss incurred by down-scaling and rounding operations. The experimental result shows that the down-scaling approach with $F(4\times4,3\times3)$ drops the model accuracy to zero, which is not acceptable. Conversely, our approach can maintain the accuracy at an acceptable level as the original model for both $F(2\times2,3\times3)$ and $F(4\times4,3\times3)$.

Table 4 shows the speedups of end-to-end model inference with different INT8 Winograd convolutions, including oneDNN's INT8 Winograd convolution with $F(2\times2,3\times3)$, LoWino's Winograd convolutions with $F(2\times2,3\times3)$ and $F(4\times4,3\times3)$. Compared with oneDNN's $F(2\times2,3\times3)$ INT8 Winograd convolutions, LoWino achieves 1.34× and 2.04× for VGG16 and 1.05 × and 1.11 × for ResNet-50, with $F(2\times2,3\times3)$ and $F(4\times4,3\times3)$, respectively. We observed that the VGG16 model achieves higher speedup than ResNet-50, which is due to the proportion of $3\times3$ convolution in VGG16 is higher than ResNet-50. The most of layers in VGG16 are $3\times3$ convolutions, whereas the residual structures in ResNet-50 have both $1\times1$ convolutions and $3\times3$ convolutions.

For the small tile size, both in-side quantization and out-side quantization achieve tolerable accuracy loss. Meanwhile, their performances are at a similar level, which is because both of them employ carefully designed optimization. However, for the large tile sizes, the out-side quantization with the down-scaling approach drops the model accuracy to zero, which is not acceptable, whereas our in-side approach can maintain the accuracy at an acceptable level and achieve higher performance. These results validate the effectiveness of LoWino (using inside quantization), which can outperform existing approaches in the vendor library.

### 5.3 Error Analysis

To illustrate the interference incurred by combining Winograd convolutions with quantization techniques in existing approaches, we leverage two representative deep neural networks, VGG16 [52] and ResNet-50 [18], to analyze the computational errors of convolutions. We measured the *mean absolute error* and the *relative error* of Winograd INT8 convolutional layers with filter size $3\times3$ when using the down-scaling approach. The ground truth is estimated by using low-precision direct convolution with INT8 arithmetic. Let $\mathcal{Y}^*$ be a result matrix (INT8 Winograd convolutions) and $\mathcal{Y}$ be the ground truth (INT8 direct convolutions). The values of input tensors are randomly generated with the Normal distribution $N(0,1)$ and then quantized to the INT8 low-precision type, while the values of filter tensors are initialized with the pre-trained models [18, 52] and then quantized to the INT8 low-precision type. The mean absolute error $E_{\mathtt{abs}}$ is defined as the absolute of the average difference between matrices $\mathcal{Y}$ and $\mathcal{Y}^*$. The relative error in the Frobenius norm $E_{\mathtt{rel}}$ (relative to the data range of $\mathcal{Y}^*$) is defined as:

$$E_{\mathtt{rel}}(\mathcal{Y},\mathcal{Y}^*) \quad = \quad \frac{\|\mathcal{Y}-\mathcal{Y}^*\|_F}{\|\mathcal{Y}^*\|_F} \tag{11}$$

Table 5 shows the computational errors ($E_{\mathtt{abs}}$ and $E_{\mathtt{rel}}$) measured with different low-precision Winograd convolution approaches, including the down-scaling approach and the approach proposed in this paper (i.e., LoWino). The existing down-scaling approach is only appropriate for small tile sizes, such as $F(2\times2,3\times3)$, due to the dramatically increasing computational errors with large tile sizes. However, the Winograd convolution with larger tile sizes can reduce more computations and has more potential for realistic acceleration. Table 5 shows that our approach can significantly

Table 5. Absolute and relative errors of representative CNNs with different approaches.

| Model | INT8 Winograd Convolution F(2×2, 3×3) | | | | INT8 Winograd Convolution F(4×4, 3×3) | | | |
|---|---|---|---|---|---|---|---|---|
| | Down-Scaling Approach | | LoWino (Ours) | | Down-Scaling Approach | | LoWino (Ours) | |
| | $E_{\mathtt{abs}}$ | $E_{\mathtt{rel}}$ | $E_{\mathtt{abs}}$ | $E_{\mathtt{rel}}$ | $E_{\mathtt{abs}}$ | $E_{\mathtt{rel}}$ | $E_{\mathtt{abs}}$ | $E_{\mathtt{rel}}$ |
| VGG16 | 1.525E-01 | 1.664E-01 | 7.892E-02 | 8.636E-02 | 1.929E-00 | 2.314E-00 | 8.076E-01 | 9.214E-01 |
| ResNet-50 | 8.147E-02 | 1.391E-01 | 5.209E-02 | 8.861E-02 | 1.147E-00 | 2.158E-00 | 5.086E-01 | 9.003E-01 |

Table 6. Comparing $E_{\mathtt{abs}}$ with different $F(2\times 2, 3\times 3)$ low-precision Winograd approaches.

| **Method** | **H&W** | **(C, K)** | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | **(64, 64)** | **(128, 128)** | **(256, 256)** | **(256, 512)** | **(512, 512)** | **(512, 1024)** | **(1024, 1024)** |
| D-S | 8 | 5.794E-01 | 7.897E-01 | 1.224E+00 | 1.190E+00 | 1.684E+00 | 1.756E+00 | 2.528E+00 |
| LoWino | | 3.286E-01 | 4.878E-01 | 6.774E-01 | 7.225E-01 | 1.020E+00 | 1.012E+00 | 1.502E+00 |
| Reduction | | 43.28% | 38.23% | 44.64% | 39.30% | 39.46% | 42.34% | 40.60% |
| D-S | 16 | 6.029E-01 | 8.416E-01 | 1.232E+00 | 1.247E+00 | 1.914E+00 | 1.808E+00 | 2.647E+00 |
| LoWino | | 3.919E-01 | 5.208E-01 | 7.448E-01 | 7.592E-01 | 1.126E+00 | 1.088E+00 | 1.606E+00 |
| Reduction | | 35.00% | 38.11% | 39.55% | 39.10% | 41.15% | 39.82% | 39.32% |
| D-S | 32 | 6.408E-01 | 9.203E-01 | 1.395E+00 | 1.391E+00 | 1.904E+00 | 1.975E+00 | 2.779E+00 |
| LoWino | | 3.634E-01 | 5.227E-01 | 7.584E-01 | 7.607E-01 | 1.096E+00 | 1.138E+00 | 1.591E+00 |
| Reduction | | 43.28% | 43.20% | 45.64% | 45.33% | 42.40% | 42.37% | 42.75% |
| D-S | 64 | 6.826E-01 | 9.441E-01 | 1.390E+00 | 1.393E+00 | 1.953E+00 | 1.972E+00 | 2.805E+00 |
| LoWino | | 3.821E-01 | 5.427E-01 | 8.473E-01 | 8.091E-01 | 1.126E+00 | 1.158E+00 | 1.685E+00 |
| Reduction | | 44.03% | 42.52% | 39.05% | 41.90% | 42.37% | 41.25% | 39.93% |
| D-S | 128 | 6.943E-01 | 9.812E-01 | 1.398E+00 | 1.399E+00 | 1.978E+00 | 1.983E+00 | 2.794E+00 |
| LoWino | | 3.926E-01 | 5.560E-01 | 8.423E-01 | 8.373E-01 | 1.137E+00 | 1.176E+00 | 1.740E+00 |
| Reduction | | 43.45% | 43.34% | 39.74% | 40.16% | 42.53% | 40.72% | 37.72% |

reduce $E_{\mathtt{abs}}$ and $E_{\mathtt{rel}}$ compared with the down-scaling approach. The quantization process inside of the Winograd domain fully utilizes the range of the low-precision data type, which reduces the computational errors. Meanwhile, it performs efficient INT8 computing for the multiplication operations, which avoids the performance degradation of the up-casting approach, thereby yielding large performance improvements. The result demonstrates that LoWino is an effective mechanism to utilize larger tile sizes for low-precision Winograd convolutions, so as to further improve performance with reasonable accuracy.

We also analyze the computational errors ($E_{\mathtt{abs}}$ and $E_{\mathtt{rel}}$) of different convolution layers with various configurations (including $C$, $K$, and $H\&W$), by leveraging the down-scaling (D-S) approach and LoWino. Table 6 and Table 7 show the results of $E_{\mathtt{abs}}$, while Table 8 and Table 9 show the results of $E_{\mathtt{rel}}$. First, LoWino significantly reduces $E_{\mathtt{abs}}$ by up to 45.64% and 85.89% and reduces $E_{\mathtt{rel}}$ by up to 47.44% and 86.84%, over the down-scaling approach for $F(2\times 2, 3\times 3)$ and $F(4\times 4, 3\times 3)$, respectively. Our approach is effective for various configurations of convolution layers, which can be widely applied in intelligent applications. Second, the computational error incurred increases sharply when moving from $F(2\times 2, 3\times 3)$ to $F(4\times 4, 3\times 3)$ for low-precision Winograd convolutions, and the down-scaling approach cannot meet the accuracy requirement with large tile sizes, which is validated in Table 3. Nevertheless, benefiting our inside-quantization design, LoWino achieves lower errors than existing approaches and can accelerate convolutional neural networks under an acceptable accuracy level.

To further explain the different quantization methods, we use VGG16_a as an example to illustrate the difference between the down-scaling approach and LoWino. Figure 12 depicts the data distribution of the transformed inputs before and after scaling/quantization. The X-axis is the range of values and Y-axis is the count of each value (in the logarithmic scale). The input

Table 7. Comparing $E_{\mathtt{abs}}$ with different $F(4 \times 4, 3 \times 3)$ low-precision Winograd approaches.

| Method | H&W | (C, K) | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | **(64, 64)** | **(128, 128)** | **(256, 256)** | **(256, 512)** | **(512, 512)** | **(512, 1024)** | **(1024, 1024)** |
| D-S | 8 | 1.653E+01 | 2.275E+01 | 3.386E+01 | 3.326E+01 | 4.906E+01 | 5.054E+01 | 6.900E+01 |
| LoWino | | 2.396E+00 | 3.499E+00 | 5.026E+00 | 4.978E+00 | 7.360E+00 | 7.119E+00 | 1.066E+01 |
| Reduction | | 85.51% | 84.62% | 85.16% | 85.03% | 85.00% | 85.91% | 84.55% |
| D-S | 16 | 1.583E+01 | 2.521E+01 | 3.011E+01 | 3.220E+01 | 4.606E+01 | 4.780E+01 | 6.599E+01 |
| LoWino | | 2.421E+00 | 3.556E+00 | 5.196E+00 | 5.147E+00 | 7.207E+00 | 7.479E+00 | 1.059E+01 |
| Reduction | | 84.70% | 85.89% | 82.74% | 84.01% | 84.35% | 84.35% | 83.96% |
| D-S | 32 | 1.527E+01 | 2.257E+01 | 3.146E+01 | 3.079E+01 | 4.475E+01 | 4.534E+01 | 6.507E+01 |
| LoWino | | 2.494E+00 | 3.613E+00 | 5.056E+00 | 5.566E+00 | 7.646E+00 | 7.667E+00 | 1.081E+01 |
| Reduction | | 83.67% | 83.99% | 83.93% | 81.93% | 82.92% | 83.09% | 83.39% |
| D-S | 64 | 1.504E+01 | 2.243E+01 | 3.216E+01 | 3.100E+01 | 4.563E+01 | 4.587E+01 | 6.574E+01 |
| LoWino | | 2.570E+00 | 3.739E+00 | 5.168E+00 | 5.400E+00 | 7.650E+00 | 7.723E+00 | 1.118E+01 |
| Reduction | | 82.91% | 83.33% | 83.93% | 82.58% | 83.24% | 83.16% | 83.00% |
| D-S | 128 | 1.579E+01 | 2.208E+01 | 3.235E+01 | 3.252E+01 | 4.574E+01 | 4.590E+01 | 6.513E+01 |
| LoWino | | 2.518E+00 | 3.687E+00 | 5.442E+00 | 5.551E+00 | 7.718E+00 | 7.697E+00 | 1.143E+01 |
| Reduction | | 84.05% | 83.30% | 83.18% | 82.93% | 83.13% | 83.23% | 82.45% |

Table 8. Comparing $E_{\mathtt{rel}}$ with different $F(2 \times 2, 3 \times 3)$ low-precision Winograd approaches.

| Method | H&W | (C, K) | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | **(64, 64)** | **(128, 128)** | **(256, 256)** | **(256, 512)** | **(512, 512)** | **(512, 1024)** | **(1024, 1024)** |
| D-S | 8 | 5.232E-02 | 5.095E-02 | 5.595E-02 | 5.383E-02 | 5.390E-02 | 5.377E-02 | 5.801E-02 |
| LoWino | | 2.953E-02 | 3.171E-02 | 3.095E-02 | 3.279E-02 | 3.254E-02 | 3.116E-02 | 3.452E-02 |
| Reduction | | 43.56% | 37.76% | 44.68% | 39.09% | 39.63% | 42.05% | 40.49% |
| D-S | 16 | 5.581E-02 | 5.329E-02 | 5.765E-02 | 5.646E-02 | 5.851E-02 | 5.926E-02 | 5.993E-02 |
| LoWino | | 3.173E-02 | 3.387E-02 | 3.370E-02 | 3.316E-02 | 3.443E-02 | 3.330E-02 | 3.462E-02 |
| Reduction | | 43.15% | 36.44% | 41.54% | 41.27% | 41.16% | 43.81% | 42.23% |
| D-S | 32 | 6.259E-02 | 5.877E-02 | 6.058E-02 | 5.711E-02 | 5.913E-02 | 6.401E-02 | 6.189E-02 |
| LoWino | | 3.290E-02 | 3.381E-02 | 3.545E-02 | 3.341E-02 | 3.483E-02 | 3.529E-02 | 3.567E-02 |
| Reduction | | 47.44% | 42.47% | 41.48% | 41.50% | 41.10% | 44.87% | 42.37% |
| D-S | 64 | 6.304E-02 | 6.322E-02 | 6.118E-02 | 6.084E-02 | 6.345E-02 | 6.187E-02 | 6.203E-02 |
| LoWino | | 3.411E-02 | 3.488E-02 | 3.621E-02 | 3.526E-02 | 3.774E-02 | 3.572E-02 | 3.679E-02 |
| Reduction | | 45.89% | 44.83% | 40.81% | 42.04% | 40.52% | 42.27% | 40.69% |
| D-S | 128 | 6.189E-02 | 6.188E-02 | 6.328E-02 | 6.344E-02 | 6.338E-02 | 6.346E-02 | 6.333E-02 |
| LoWino | | 3.615E-02 | 3.558E-02 | 3.686E-02 | 3.902E-02 | 3.674E-02 | 3.745E-02 | 3.810E-02 |
| Reduction | | 41.59% | 42.50% | 41.75% | 38.49% | 42.03% | 40.99% | 39.84% |

for the down-scaling approach has already been quantized to 8-bit integers and the value range of transformed input will be increased up to 100× after Winograd transformation (as discussed in Section 2.2). To avoid the overflow of low-precision matrix multiplication, the down-scaling approach needs to be multiplied by a scaling factor, $\alpha = \frac{1}{100}$. Moreover, the down-scaled values need to be converted to integers, which introduces rounding errors. Despite that INT8 has values with a range of [-128...127], the transformed input can only be represented by the integers in a narrower range as shown in the figure. In our approach, the input and transformed input are full-precision values and the transformed input is quantized in the Winograd domain. Thus, we can fully use the values of [-128...127] to represent the original values, thereby reducing the precision loss.

Table 9. Comparing $E_{\text{rel}}$ with different $F(4\times4, 3\times3)$ low-precision Winograd approaches.

| Method | H&W | (C, K) | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | (64, 64) | (128, 128) | (256, 256) | (256, 512) | (512, 512) | (512, 1024) | (1024, 1024) |
| D-S | 8 | 1.785E+00 | 1.809E+00 | 1.864E+00 | 1.822E+00 | 1.919E+00 | 1.882E+00 | 1.938E+00 |
| LoWino | | 2.349E-01 | 2.505E-01 | 2.515E-01 | 2.481E-01 | 2.584E-01 | 2.396E-01 | 2.688E-01 |
| Reduction | | 86.84% | 86.16% | 86.51% | 86.39% | 86.53% | 87.27% | 86.13% |
| D-S | 16 | 1.741E+00 | 2.001E+00 | 1.671E+00 | 1.767E+00 | 1.786E+00 | 1.898E+00 | 1.813E+00 |
| LoWino | | 2.393E-01 | 2.515E-01 | 2.586E-01 | 2.523E-01 | 2.459E-01 | 2.661E-01 | 2.630E-01 |
| Reduction | | 86.26% | 87.43% | 84.53% | 85.73% | 86.24% | 85.98% | 85.49% |
| D-S | 32 | 1.706E+00 | 1.757E+00 | 1.753E+00 | 1.736E+00 | 1.748E+00 | 1.775E+00 | 1.814E+00 |
| LoWino | | 2.480E-01 | 2.515E-01 | 2.505E-01 | 2.810E-01 | 2.676E-01 | 2.678E-01 | 2.696E-01 |
| Reduction | | 85.46% | 85.68% | 85.71% | 83.81% | 84.69% | 84.91% | 85.14% |
| D-S | 64 | 1.684E+00 | 1.775E+00 | 1.788E+00 | 1.717E+00 | 1.784E+00 | 1.792E+00 | 1.814E+00 |
| LoWino | | 2.569E-01 | 2.641E-01 | 2.562E-01 | 2.671E-01 | 2.678E-01 | 2.701E-01 | 2.765E-01 |
| Reduction | | 84.74% | 85.12% | 85.67% | 84.44% | 84.99% | 84.93% | 84.76% |
| D-S | 128 | 1.765E+00 | 1.732E+00 | 1.793E+00 | 1.798E+00 | 1.788E+00 | 1.808E+00 | 1.815E+00 |
| LoWino | | 2.520E-01 | 2.590E-01 | 2.693E-01 | 2.760E-01 | 2.708E-01 | 2.705E-01 | 2.848E-01 |
| Reduction | | 85.72% | 85.04% | 84.98% | 84.65% | 84.86% | 85.04% | 84.31% |

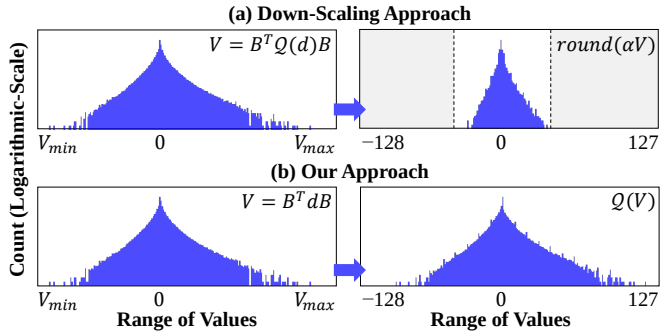



Fig. 12. Comparing the down-scaling approach with ours for $F(4\times4, 3\times3)$ low-precision Winograd convolution.

## 5.4 Performance Analysis

*5.4.1 Overhead of Quantization Operations.* We further analyze the overhead of quantization-related operations, including quantization operations in the input transformation stage and de-quantization operations in the output transformation stage. The quantization operations are responsible for converting the high-precision data to the low-precision data, and the de-quantization operations do the opposite. Figure 13 shows the results. The quantization overheads of LoWino with non-fused implementation approaches with $F(2\times2, 3\times3)$ and $F(4\times4, 3\times3)$ are 13.93% and 18% and the quantization overheads of LoWino with fused implementation are 8.23% and 8%. As the fused one improves the data reuse, the overheads of the fused implementation are lower than the non-fused implementation.

In this paper, we implemented our low-precision Winograd convolution (i.e. LoWino) on Intel CPUs based on Intel VNNI, which supports high-performance vector low-precision computation. The specific instructions of modern architectures provide higher efficiency of computations than traditional non-vector instructions. When using traditional instructions (e.g., Intel SSE), the proportion of computational parts will increase and the proportion of quantization parts will decrease, due to the lower computation efficiency of the instructions.

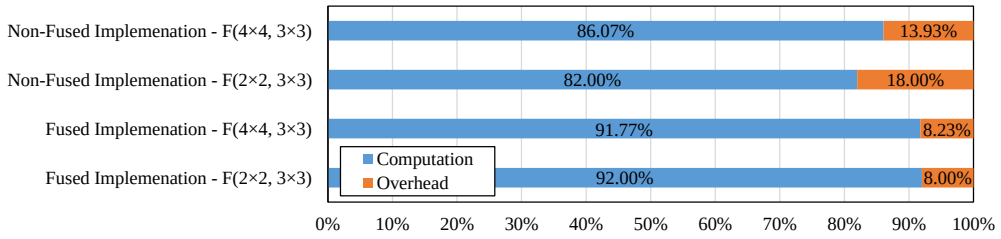



Fig. 13. Analyzing the quantization overheads of LoWino.

Table 10. Cache miss reduction of fused implementation.

| LoWino Implementation | | Cache Reference | Cache Miss | Cache Miss Rate | Reduction |
|---|---|---|---|---|---|
| $F(2\times2, 3\times3)$ | Non-Fused | 1.20E+09 | 8.55E+08 | 71.53% | 29.74% |
| | Fused | 1.66E+09 | 6.94E+08 | 41.79% | |
| $F(4\times4, 3\times3)$ | Non-Fused | 7.24E+08 | 5.40E+08 | 74.53% | 37.83% |
| | Fused | 1.27E+09 | 4.65E+08 | 36.70% | |

*5.4.2 Reduction of Cache Miss with Fused Implementation.* As introduced in Section 4.4, the fused and non-fused methods are both implemented in LoWino. The fused implementation aims to reduce the memory access overhead by reusing the data in the cache, thereby increasing the cache hit rate. We conduct experiments to measure the cache miss for non-fused and fused implementations. Table 10 shows the results, including the number of cache reference events, the number of cache miss events, the cache miss rate, and the cache miss reduction rate. The fused implementation leads to more cache reference events than the non-fused one because more data are stored in the cache, thereby reducing the cache miss rate. For LoWino with $F(2\times2, 3\times3)$, the fused implementation can reduce the cache miss rate from 71.53% to 41.79%, achieving a 29.74% reduction. Similarly, for LoWino with $F(4\times4, 3\times3)$, the cache miss rate is reduced from 74.53% to 36.70% when using fused implementation. The result validates that our fused implementation can effectively reduce the cache miss and mitigate memory overheads, so as to improve the performance.

## 6 DISCUSSION

**Generality and Applicability.** Convolution operators arguably dominate the computation of modern CNNs, which usually use small, 3×3 filters. Our approach focuses on optimizing 3×3 convolutions by combining the Winograd algorithm with low-precision computing. We have demonstrated the effectiveness of LoWino, which can be applied to various convolutional neural network models. Moreover, LoWino can be potentially effective for many other emerging neural network models such as CMT [15].

**Performance-Precision Tradeoffs.** It is a long-standing research problem to make trade-offs between performance and precision for intelligent applications. LoWino is designed for accelerating CNN applications while meeting the accuracy requirement. We have analyzed the computational errors incurred by combining fast convolution with low-precision quantization techniques and explored how to design efficient quantized Winograd convolution. To determine fine-grained settings for each operator that achieves a suitable performance-accuracy trade-off for a given neural network, an auto-tuning method will be developed in future work.

**Low-Precision Quantization.** When developing LoWino, we employed a symmetric linear quantization method, which has low overhead and can be effectively accelerated on modern CPUs. We

utilize a calibration technique to avoid the time-consuming process of model re-training. Complicated quantization schemes, such as non-linear quantization [3], may further reduce the precision loss. In addition, the numeric formats with shared or dynamic exponent, such as hybrid block floating point [11], flexible floating-point [30], and FP8 [44], provide low-precision floating-point computation capacity. Exploring different quantization schemes and new low-precision data types for low-precision quantized Winograd convolutions is another future research.

**Hardware Platforms.** In this paper, we focus on exploring efficient quantized Winograd convolution on CPUs with VNNI. The CPU is one of the most ubiquitous resources in high-performance computing platforms (e.g., data centers and cloud servers) and embedded computing platforms (mobile phones and robots). In recent years, there have been many works focused on optimizing DNN inference on CPU platforms [4, 9, 14]. Nevertheless, the in-side quantization methodology of LoWino is hardware-independent, which can be easily applied to other hardware platforms. In future work, we will explore implementing LoWino on other platforms such as GPUs and NPUs.

## 7 RELATED WORK

Winograd's minimal filtering algorithm is introduced by Lavin and Gray [36] to reduce the complexity of compute-intensive convolutions in CNNs. Afterward, the Winograd-based convolution was integrated into popular vendor libraries and has been widely used to accelerate convolutional neural networks for its proven effectiveness. Recently, many efforts have been made to optimize the Winograd convolutions on CPU platforms [28, 38] and GPU platforms [26, 43, 59]. Jia *et al.* [28] proposed an implementation to support n-dimensional Winograd-based convolutions on many-core CPUs. Li *et al.* [38] optimized massively parallel Winograd convolution on ARM platforms. Xie *et al.* [58] explored efficient half-precision Winograd convolution on ARM many-core processors. Besides, Huang *et al.* [21] presented a decomposable Winograd method that supports convolution layers with large kernels and large strides. Andri *et al.* [1] designed a tap-wise quantization method for integer-only inference and customized Winograd-enhanced accelerators. To optimize Winograd convolutions on GPUs, Mazaheri *et al.* [43] proposed an efficient method via symbolic computation and meta-programming, and Yan *et al.* [59] built an assembler to perform assembly-level optimizations. Liu *et al.* [41] accelerated Winograd convolution on GPU Tensor Cores with mixed-precision computing, where the matrix multiplications are performed under FP16. Combining the Winograd algorithm with low-precision computation is a growing trend for convolutional neural network acceleration on emerging platforms. However, the outside quantization method causes the precision issue and limits the potential speedups, whereas the inside quantization methodology of LoWino resolves the problem. In general, benefiting from the Winograd algorithm demands system-level optimizations of input/output transformations and multiplication operations, and the operations related to low-precision computation, such as quantization and de-quantization, introduce extra complexity. In this paper, we focus on exploring efficient quantized Winograd convolution on the CPU, which is one of the most ubiquitous resources in high-performance computing platforms (e.g., data centers and cloud servers) and embedded computing platforms (mobile phones and robots). Nevertheless, the inside quantization methodology of LoWino is hardware-independent, which can be easily applied to other hardware platforms. In future work, we will explore implementing LoWino on more platforms such as GPUs.

Some efforts have been made to exploit the integer-arithmetic computations by converting the full-precision models into low-precision [6, 25, 32, 45, 47, 61], and further improvement of low-precision quantization techniques for Winograd convolutions are also expected [40, 56]. An up-casting approach solution is adopted by ncnn [54] for low-precision Winograd convolution, and a down-scaling approach is provided by oneDNN [24]. As demonstrated in Section 2.3, these

traditional methods applied in vendor libraries still suffer from accuracy loss or performance degradation. Meanwhile, some researchers notice the disadvantages and try to utilize searching and re-training on the training dataset [12, 39]. Those methods rely on re-training to reduce the accuracy loss introduced by quantization and Winograd convolutions, whereas our approach reduces the accuracy loss by performing the inside-quantization in the Winograd domain without re-training. Our approach provides an effective mechanism for combining Winograd convolution with low-precision computations, making it possible to support versatile problem sizes, thereby unleashing the potential of low-precision fast convolutions.

## 8 CONCLUSION

In this paper, we designed an efficient quantized Winograd convolution approach, LoWino, which employs inside-quantization in the transformed domain, effectively reducing the precision loss and exploiting the capability of low-precision computing on modern CPUs. Evaluation with state-of-the-art CNNs demonstrates that LoWino achieves up to 2.90× speedup over the best implementations in the existing vendor library. In future work, we would like to combine our approach with graph-level optimizations, such as TASO [27], to further improve the performance.

## ACKNOWLEDGMENTS

Guangli Li and Xiaobing Feng are grateful for the funding support from the National Key R&D Program of China (2021ZD0110101), the National Natural Science Foundation of China (62090024, 62232015, 62302479), the China Postdoctoral Science Foundation (2023M733566), and the Innovation Funding of ICT, CAS (E361010).

## REFERENCES

[1] Renzo Andri, Beatrice Bussolino, Antonio Cipolletta, Lukas Cavigelli, and Zhe Wang. 2022. Going Further With Winograd Convolutions: Tap-Wise Quantization for Efficient Inference on 4x4 Tiles. In *International Symposium on Microarchitecture*. IEEE, 582–598.

[2] Barbara Barabasz, Andrew Anderson, Kirk M Soodhalter, and David Gregg. 2020. Error analysis and improving the accuracy of Winograd convolution for deep neural networks. *ACM Trans. Math. Software* 46, 4 (2020), 1–33.

[3] Zhaowei Cai, Xiaodong He, Jian Sun, and Nuno Vasconcelos. 2017. Deep learning with low precision by half-wave gaussian quantization. In *IEEE conference on computer vision and pattern recognition*. 5918–5926.

[4] Beidi Chen, Tharun Medini, James Farwell, Charlie Tai, Anshumali Shrivastava, et al. 2020. Slide: In defense of smart algorithms over hardware acceleration for large-scale deep learning systems. *Proceedings of Machine Learning and Systems* 2, 1 (2020), 291–306.

[5] Jian Cheng, Pei-song Wang, Gang Li, Qing-hao Hu, and Han-qing Lu. 2018. Recent advances in efficient computation of deep convolutional neural networks. *Frontiers of Information Technology & Electronic Engineering* 19, 1 (2018), 64–77.

[6] Vladimir Chikin and Vladimir Kryzhanovskiy. 2022. Channel Balancing for Accurate Quantization of Winograd Convolutions. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 12507–12516.

[7] Yoni Choukroun, Eli Kravchik, Fan Yang, and Pavel Kisilev. 2019. Low-bit Quantization of Neural Networks for Efficient Inference.. In *ICCV Workshops*. 3009–3018.

[8] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2016. Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. *arXiv preprint arXiv:1602.02830* (2016).

[9] Shabnam Daghaghi, Nicholas Meisburger, Mengnan Zhao, and Anshumali Shrivastava. 2021. Accelerating slide deep learning on modern cpus: Vectorization, quantizations, memory optimizations, and more. *Proceedings of Machine Learning and Systems* 3 (2021), 156–166.

[10] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. In *IEEE conference on computer vision and pattern recognition*. Ieee, 248–255.

[11] Mario Drumond, Tao Lin, Martin Jaggi, and Babak Falsafi. 2018. Training dnns with hybrid block floating point. *Advances in Neural Information Processing Systems* 31 (2018).

[12] Javier Fernández-Marqués, Paul N. Whatmough, Andrew Mundy, and Matthew Mattina. 2020. Searching for Winograd-aware Quantized Networks. In *Proceedings of Machine Learning and Systems*. 1–16.

[13] Li Gaungli, Zhen Jia, Xiaobing Feng, and Yida Wang. 2021. LoWino: Towards Efficient Low-Precision Winograd Convolutions on Modern CPUs. In *International Conference on Parallel Processing*. 1–11.
[14] Zhangxiaowen Gong, Houxiang Ji, Christopher W Fletcher, Christopher J Hughes, Sara Baghsorkhi, and Josep Torrellas. 2020. Save: Sparsity-aware vector engine for accelerating dnn training and inference on cpus. In *International Symposium on Microarchitecture*. IEEE, 796–810.
[15] Jianyuan Guo, Kai Han, Han Wu, Yehui Tang, Xinghao Chen, Yunhe Wang, and Chang Xu. 2022. CMT: Convolutional Neural Networks Meet Vision Transformers. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 12175–12185.
[16] Yunhui Guo. 2018. A survey on methods and theories of quantized neural networks. *arXiv preprint arXiv:1808.04752* (2018).
[17] Song Han, Huizi Mao, and William J Dally. 2015. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149* (2015).
[18] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *IEEE Conference on Computer Vision and Pattern Recognition*. 770–778.
[19] Yang He, Xuanyi Dong, Guoliang Kang, Yanwei Fu, Chenggang Yan, and Yi Yang. 2019. Asymptotic soft filter pruning for deep convolutional neural networks. *IEEE transactions on cybernetics* 50, 8 (2019), 3594–3604.
[20] Yihui He, Xiangyu Zhang, and Jian Sun. 2017. Channel pruning for accelerating very deep neural networks. In *IEEE international conference on computer vision*. 1389–1397.
[21] Di Huang, Xishan Zhang, Rui Zhang, Tian Zhi, Deyuan He, Jiaming Guo, Chang Liu, Qi Guo, Zidong Du, Shaoli Liu, et al. 2020. DWM: a decomposable winograd method for convolution acceleration. In *AAAI Conference on Artificial Intelligence*, Vol. 34. 4174–4181.
[22] Intel. 2021. *Intrinsics Guide*. Retrieved March 29, 2021 from https://software.intel.com/sites/landingpage/IntrinsicsGuide/
[23] Intel. 2021. *Introduction to Intel Deep Learning Boost on Second Generation Intel Xeon Scalable Processors*. Retrieved March 24, 2021 from https://software.intel.com/content/www/us/en/develop/articles/introduction-to-intel-deep-learning-boost-on-second-generation-intel-xeon-scalable.html
[24] Intel. 2021. *oneAPI Deep Neural Network Library (oneDNN)*. Retrieved February 27, 2021 from https://github.com/oneapi-src/oneDNN
[25] 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 *IEEE Conference on Computer Vision and Pattern Recognition*. 2704–2713.
[26] Liancheng Jia, Yun Liang, Xiuhong Li, Liqiang Lu, and Shengen Yan. 2020. Enabling efficient fast convolution algorithms on GPUs via MegaKernels. *IEEE Trans. Comput.* 69, 7 (2020), 986–997.
[27] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. 2019. TASO: optimizing deep learning computation with automatic generation of graph substitutions. In *ACM Symposium on Operating Systems Principles*. 47–62.
[28] Zhen Jia, Aleksandar Zlateski, Fredo Durand, and Kai Li. 2018. Optimizing N-dimensional, winograd-based convolution for manycore CPUs. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 109–123.
[29] Zhen Jia, Aleksandar Zlateski, Fredo Durand, and Kai Li. 2018. Towards Optimal Winograd Convolution on Manycores. *Proceedings of Machine Learning and Systems*, 1–3.
[30] Urs Köster, Tristan Webb, Xin Wang, Marcel Nassar, Arjun K Bansal, William Constable, Oguz Elibol, Scott Gray, Stewart Hall, Luke Hornof, et al. 2017. Flexpoint: An adaptive numerical format for efficient training of deep neural networks. *Advances in neural information processing systems* 30 (2017).
[31] Moez Krichen. 2023. Convolutional neural networks: A survey. *Computers* 12, 8 (2023), 151.
[32] Raghuraman Krishnamoorthi. 2018. Quantizing deep convolutional networks for efficient inference: A whitepaper. *arXiv preprint arXiv:1806.08342* (2018).
[33] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. *Advances in Neural Information Processing Systems* 25 (2012), 1097–1105.
[34] Solomon Kullback. 1997. *Information theory and statistics*. Courier Corporation.
[35] Andrew Lavin. 2021. *wincnn*. Retrieved February 27, 2021 from https://github.com/andravin/wincnn
[36] Andrew Lavin and Scott Gray. 2016. Fast algorithms for convolutional neural networks. In *IEEE Conference on Computer Vision and Pattern Recognition*. 4013–4021.
[37] Chendi Li, Haipeng Jia, Hang Cao, Jianyu Yao, Boqian Shi, Chunyang Xiang, Jinbo Sun, Pengqi Lu, and Yunquan Zhang. 2021. Autotsmm: An auto-tuning framework for building high-performance tall-and-skinny matrix-matrix multiplication on cpus. In *IEEE Intl Conf on Parallel & Distributed Processing with Applications*. IEEE, 159–166.
[38] Dongsheng Li, Dan Huang, Zhiguang Chen, and Yutong Lu. 2021. Optimizing Massively Parallel Winograd Convolution on ARM Processor. In *International Conference on Parallel Processing*. 1–12.

[39] Guangli Li, Lei Liu, Xueying Wang, Xiu Ma, and Xiaobing Feng. 2020. Lance: efficient low-precision quantized winograd convolution for neural networks based on graphics processing units. In *International Conference on Acoustics, Speech and Signal Processing*. IEEE, 3842–3846.

[40] Guangli Li, Jingling Xue, Lei Liu, Xueying Wang, Xiu Ma, Xiao Dong, Jiansong Li, and Xiaobing Feng. 2021. Unleashing the Low-Precision Computation Potential of Tensor Cores on GPUs. In *International Symposium on Code Generation and Optimization*. IEEE, 90–102.

[41] Junhong Liu, Dongxu Yang, and Junjie Lai. 2021. Optimizing Winograd-based convolution with tensor cores. In *International Conference on Parallel Processing*. 1–10.

[42] Yizhi Liu, Yao Wang, Ruofei Yu, Mu Li, Vin Sharma, and Yida Wang. 2019. Optimizing CNN model inference on cpus. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 1025–1040.

[43] Arya Mazaheri, Tim Beringer, Matthew Moskewicz, Felix Wolf, and Ali Jannesari. 2020. Accelerating winograd convolutions using symbolic computation and meta-programming. In *European Conference on Computer Systems*. 1–14.

[44] Paulius Micikevicius, Dusan Stosic, Neil Burgess, Marius Cornea, Pradeep Dubey, Richard Grisenthwaite, Sangwon Ha, Alexander Heinecke, Patrick Judd, John Kamalu, Naveen Mellempudi, Stuart Oberman, Mohammad Shoeybi, Michael Siu, and Hao Wu. 2022. FP8 Formats for Deep Learning. arXiv:2209.05433 [cs.LG]

[45] Szymon Migacz. 2017. 8-bit inference with tensorrt. In *GPU technology conference*, Vol. 2. 5.

[46] NVIDIA. 2021. *CUDA C++ Programming Guide*. Retrieved March 29, 2021 from https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[47] Eunhyeok Park, Sungjoo Yoo, and Peter Vajda. 2018. Value-Aware Quantization for Training and Inference of Neural Networks. In *Computer Vision - ECCV 2018 - 15th European Conference*. 608–624.

[48] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems*. 8024–8035.

[49] Tran Minh Quan, David Grant Colburn Hildebrand, and Won-Ki Jeong. 2021. Fusionnet: A deep fully residual convolutional neural network for image segmentation in connectomics. *Frontiers in Computer Science* 3 (2021), 613981.

[50] Joseph Redmon and Ali Farhadi. 2018. Yolov3: An incremental improvement. *arXiv preprint arXiv:1804.02767* (2018).

[51] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. 2015. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical Image Computing and Computer-Assisted Intervention*. Springer, 234–241.

[52] Karen Simonyan and Andrew Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *International Conference on Learning Representations*. 1–14.

[53] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going deeper with convolutions. In *IEEE Conference on Computer Vision and Pattern Recognition*. 1–9.

[54] Tencent. 2021. *ncnn*. Retrieved February 27, 2021 from https://github.com/Tencent/ncnn

[55] Yida Wang, Michael J Anderson, Jonathan D Cohen, Alexander Heinecke, Kai Li, Nadathur Satish, Narayanan Sundaram, Nicholas B Turk-Browne, and Theodore L Willke. 2015. Full correlation matrix analysis of fMRI data on Intel® Xeon Phi™ coprocessors. In *International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–12.

[56] Jian Weng, Animesh Jain, Jie Wang, Leyuan Wang, Yida Wang, and Tony Nowatzki. 2021. UNIT: Unifying Tensorized Instruction Compilation. In *International Symposium on Code Generation and Optimization*. IEEE, 77–89.

[57] Shmuel Winograd. 1980. *Arithmetic complexity of computations*. Vol. 33. Siam.

[58] Dedong Xie, Zhen Jia, Zili Zhang, and Xin Jin. 2022. Optimizing half precision Winograd convolution on ARM many-core processors. In *ACM SIGOPS Asia-Pacific Workshop on Systems*. 53–60.

[59] Da Yan, Wei Wang, and Xiaowen Chu. 2020. Optimizing batched winograd convolution on GPUs. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 32–44.

[60] Haojin Yang, Martin Fritzsche, Christian Bartz, and Christoph Meinel. 2017. Bmxnet: An open-source binary neural network implementation based on mxnet. In *International Conference on Multimedia*. 1209–1212.

[61] Yiwu Yao, Bin Dong, Yuke Li, Weiqiang Yang, and Haoqi Zhu. 2019. Efficient implementation of convolutional neural networks with end to end integer-only dataflow. In *IEEE International Conference on Multimedia and Expo*. 1780–1785.

[62] Zhewei Yao, Zhen Dong, Zhangcheng Zheng, Amir Gholami, Jiali Yu, Eric Tan, Leyuan Wang, Qijing Huang, Yida Wang, Michael Mahoney, et al. 2021. Hawq-v3: Dyadic neural network quantization. In *International Conference on Machine Learning*. PMLR, 11875–11886.

[63] Aleksandar Zlateski, Zhen Jia, Kai Li, and Fredo Durand. 2019. The anatomy of efficient FFT and winograd convolutions on modern CPUs. In *International Conference on Supercomputing*. 414–424.