LoWino: Towards Efficient Low-Precision Winograd Convolutions on Modern CPUs

Guangli Li*
[1]SKLCA, Institute of Computing Technology, Chinese Academy of Sciences
Beijing, China
[2]University of Chinese Academy of Sciences
Beijing, China
liguangli@ict.ac.cn

Zhen Jia
Amazon Web Services
East Palo Alto, CA, USA
zhej@amazon.com

Xiaobing Feng
[1]SKLCA, Institute of Computing Technology, Chinese Academy of Sciences
Beijing, China
[2]University of Chinese Academy of Sciences
Beijing, China
fxb@ict.ac.cn

Yida Wang
Amazon Web Services
East Palo Alto, CA, USA
wangyida@amazon.com

ABSTRACT

Low-precision computation, which has been widely supported in contemporary hardware, is considered as one of the most effective methods to accelerate convolutional neural networks. However, low-precision computation is not widely used to speed up Winograd, an algorithm for fast convolution computation, due to the numerical error introduced by combining Winograd transformation and quantization. In this paper, we propose a low-precision Winograd convolution approach, LoWino, based on post-training quantization, which employs a linear quantization method in the Winograd domain to reduce the precision loss caused by transformations. Moreover, we present an efficient implementation that integrates well-designed optimization techniques, thereby adequately exploiting the capability of low-precision computation on modern CPUs. We evaluate our approach on Intel Xeon Scalable Processors by leveraging representative convolutional layers in prevailing deep neural networks. Experimental results show that LoWino achieves up to 2.04× speedup over state-of-the-art implementations in the vendor library while maintaining the accuracy at a reasonable level.

CCS CONCEPTS

• **Computing methodologies** → **Neural networks**; • **Software and its engineering** → *Just-in-time compilers.*

*Work done during Guangli's internship at AWS.

ICPP '21, August 9–12, 2021, Lemont, IL, USA

ACM ISBN 978-1-4503-9068-2/21/08…$15.00
https://doi.org/10.1145/3472456.3472464

KEYWORDS

Deep Neural Networks, Low-Precision Computing, Winograd Convolution, Post-Training Quantization

ACM Reference Format:
Guangli Li, Zhen Jia, Xiaobing Feng, and Yida Wang. 2021. LoWino: Towards Efficient Low-Precision Winograd Convolutions on Modern CPUs. In *50th International Conference on Parallel Processing (ICPP '21), August 9–12, 2021, Lemont, IL, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3472456.3472464

1 INTRODUCTION

Convolutional neural networks (CNNs) have been widely used in various intelligent applications, including image classification [20], object detection [34], and semantic segmentation [35]. The superior performance of contemporary CNNs usually comes from sophisticated structures [9, 36, 37], causing an expensive computational cost. The convolutional layer, arguably the most time-consuming component of state-of-the-art CNNs, is the key to improve end-to-end performance. While many efforts have been made in fast convolution algorithms [23], performance optimization [26], model compression [8], and hardware accelerators [2], among others, the mechanism for effectively optimizing convolution is still a long-standing challenge in both industry and academia.

With the ever-increasing demand for hardware resource execution and energy efficiency, low-precision neural networks are demonstrated to be feasible solutions with little accuracy loss [3] and become an inevitable trend for optimizing convolutional neural networks. To cater the trend, chip vendors also design and introduce low-precision computation instructions in ISAs, such as VNNI on x86 architecture [12] and WMMA on Nvidia GPUs [30], which are supposed to deliver performance improvements over full-precision computation but at the same time challenge the developers who want to take fully advantage of those instructions.

On the other hand, fast convolution algorithms, represented by Winograd's minimal filtering algorithm [23], are frequently employed to accelerate convolutional layers in recent years [17, 28, 42].

Winograd algorithm can reduce the theoretical computational complexity by transforming data to the Winograd domain, similar to FFT [27], and performing computation there. The Winograd algorithm is applied to tiles of input images and kernels. Theoretically, the larger the tile size the more computations can be reduced. Whereas, due to the Winograd algorithm's numerical instability, larger tile sizes introduce more numeric errors [1, 41]. This condition exacerbates when the algorithm is deployed with low-precision arithmetic, harming the model accuracy and challenging the design of low-precision Winograd convolutions. This also indicates that Winograd convolution and quantization are not orthogonal optimization methods that can be simply combined together. A holistic design is needed to achieve both good performance and reasonable accuracy. We investigated existing Winograd convolution implementations [13, 38] and found that only one small tile size is supported for low-precision computation. The small tile size Winograd convolution can only reduce limited computations in theory and sometimes is slower than direct convolution since the extra memory overhead amortizes the benefits of computation savings, which restricts the potential performance improvements and also motivates us to investigate whether a larger tile size is possible for low-precision Winograd convolutions.

In this paper, we propose an efficient and effective low-precision Winograd convolution approach named LoWino. Different from existing methods performing quantization in the spatial domain [13, 38], we employ a linear quantization approach in the Winograd domain to reduce the information loss incurred by low-precision Winograd transformations. Our quantization methodology solves the problem that only a small tile size is feasible for low-precision Winograd convolution and owns better accuracy than existing solutions. We employ a customized data layout and design an efficient implementation for low-precision Winograd convolutions. We combine Winograd computation and quantization together, optimize the performance of the system as a whole, and employ static scheduling to achieve load-balanced multi-core parallelization. To demonstrate our approach's effectiveness and efficiency, we implemented LoWino on Intel platforms by utilizing VNNI [12] as the vehicle. However, our quantization methodology and optimization strategies are not restricted to VNNI and can be generally applied to other architectures. Evaluation with representative convolutional layers from prevailing CNNs shows that LoWino achieves remarkable speedups compared with the 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. Our contributions are summarized as follows:

- We propose a low-precision Winograd convolution approach to accelerate deep convolutional neural networks by exploiting the low-precision computing capability of modern CPUs. Our approach utilizes a post-training quantization scheme, which performs quantization in the Winograd domain, and demonstrates that larger tile size low-precision Winograd convolution can achieve reasonable accuracy.
- We present an efficient implementation of low-precision Winograd convolutions on modern CPU platforms, which employs several well-designed optimization techniques to enhance the efficiency in both computation and memory access.
- We evaluate LoWino with representative convolutional layers from prevailing CNNs to show that our approach outperforms the state-of-the-art methods, achieving an up to 2.04× speedup and an average of 1.26× speedup compared to the best implementations in the vendor library.

2 BACKGROUND AND MOTIVATION

2.1 Low-Precision Computation

Prior efforts [14, 31, 44] have demonstrated that deep neural networks can reduce both the memory footprint and computational cost via low-precision computation while resulting in only trivial or no accuracy loss. To support low-precision computation, new instructions have been introduced by hardware vendors in recent years. Intel VNNI [12] is a representative extension instruction set that is designed to accelerate deep learning for 8-bit computations on x86 architectures. Figure 1 illustrates the semantic of the 512-bit wide fused-multiply-add (FMA) instruction for 8-bit low-precision computations, i.e., `vpdpbusd`, which theoretically delivers 4× peak performance over FP32 operations.

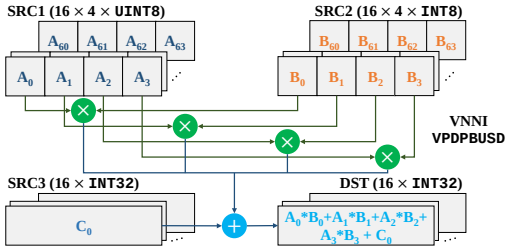


Figure 1: The semantic of the vpdpbusd instruction.

The source operands are three 512-bit registers: A, B, and C. A contains 64 unsigned 8-bit integers (UINT8), B contains 64 signed 8-bit integers (INT8), and C includes 16 signed 32-bit integers (INT32). The results are 16 32-bit integers and stored in a register D. The `vpdpbusd` performs the vector dot product between two 4-element vectors, $A_{[i\times4:i\times4+3]}$ and $B_{[i\times4:i\times4+3]}$, resulting in an 32-bit integer. The dot product result is added to the corresponding 32-bit element of C_i and accumulated to D_i. As the computation pattern of low-precision computation instructions is different from the general vector instructions, to adequately enjoy the benefits of the low-precision computation instructions, it is imperative to take its characteristics into account [40].

2.2 Winograd Algorithm

Winograd algorithm [23] is a representative fast convolution method, which reduces the theoretical computational complexity for convolution layers. For a convolutional layer performing operations on filter g and input image d with C input channels and K out channels, the Winograd convolution is represented as:

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

A, B, and G denote the transformation matrices, and $\odot$ denotes the element-wise multiplication. The $\boldsymbol{g}_{k,c}$ denotes the c-th channel of

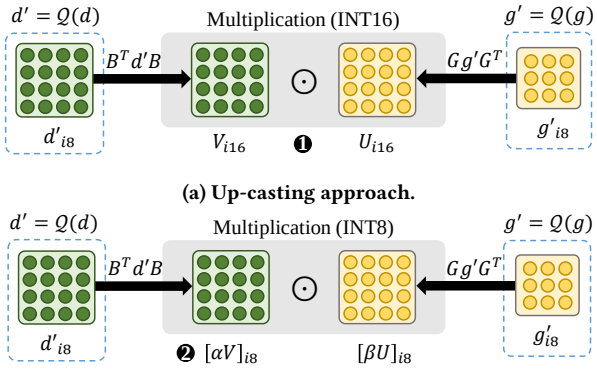
the k-th filter, $\boldsymbol{d}_c$ denotes the c-th input image, and $\boldsymbol{y}_k$ denotes the k-th channel of the output result. We follow the convention in [23] and use $F(m \times m, r \times r)$ to represent a 2D Winograd convolution, which takes an input tile size of $(m+r-1)\times(m+r-1)$ 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 of a convolutional layer is fixed whereas the output tile size m can be freely chosen. Theoretically, in the Winograd algorithm, the larger the m the more computational operations can be saved. However, due to the Winograd algorithm's numerical instability [1], the larger the m, the more numeric errors in the final result. The image sizes in modern convolutional neural networks are beyond the range Winograd algorithm can produce reasonable results. Thus, input images are usually divided into tiles with an overlap of $r-1$ along each dimension to apply the Winograd algorithm for each tile. The most frequently used Winograd algorithm for floating-point numbers is either $F(2\times2, 3\times3)$ or $F(4\times4, 3\times3)$. The transformation matrices for a specific tile size are generated by the Chinese remainder theorem [41]. As an example, the commonly used input transformation matrices [23] for Winograd convolutions are as follows:

$$B^{\mathrm{T}}_{\langle 2,3\rangle} = \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix}, B^{\mathrm{T}}_{\langle 4,3\rangle} = \begin{bmatrix} 4 & 0 & -5 & 0 & 1 & 0 \\ 0 & -4 & -4 & 1 & 1 & 0 \\ 0 & 4 & -4 & -1 & 1 & 0 \\ 0 & -2 & -1 & 2 & 1 & 0 \\ 0 & 2 & -1 & -2 & 1 & 0 \\ 0 & 4 & 0 & -5 & 0 & 1 \end{bmatrix} \quad (2)$$

$B^{\mathrm{T}}_{\langle 2,3\rangle}$ and $B^{\mathrm{T}}_{\langle 4,3\rangle}$ are input transformation matrices for the tile sizes $m=2$ and $m=4$ when the filter size $r=3$. One observation is that the transformation operations increase the range of original matrices. According to the coefficients in B^{T}, the values of the transformed input matrix will increase up to 4× and 100× after performing $B^{\mathrm{T}}\boldsymbol{d}B$ for $F(2\times2, 3\times3)$ and $F(4\times4, 3\times3)$. The coefficients of transformation matrices dramatically increase when the tile size increases [23]. This phenomenon potentially incurs numerical overflow when performing low-precision Winograd convolutions. Therefore, combining the Winograd algorithm with low-precision quantization directly may cause a non-negligible accuracy degradation. It is a challenging task to effectively implement low-precision Winograd convolutions.

2.3 Existing Approaches

As a motivating example, we analyze the state-of-the-art approaches for implementing low-precision Winograd convolutions, as shown in Figure 2. For clarity, we focus on the processes that cause precision issues, including input transformation, filter transformation, and matrix multiplication, while other parts are omitted. We assume that the problem size is $F(2\times2, 3\times3)$ and the corresponding input transformation matrix has a shape of 4×4. A straightforward approach for combining Winograd convolution with low-precision computation is to directly replace the full-precision input matrices with low-precision ones. Unfortunately, as aforementioned, the transformation operations increase the value range of the matrices, which incurs numerical overflow and thus seriously disturbs the result. Currently, there are two representative methods in vendor libraries to implement low-precision Winograd convolutions: the up-casting approach in ncnn [38] and the down-scaling approach in oneDNN [13].



(a) Up-casting approach.

(b) Down-scaling approach.

Figure 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 gray rounded rectangle represents the multiplication operation.

- **Up-Casting Approach.** Up-casting the data type of transformed matrices is a possible solution to avoid the overflow incurred by transformation operations, which up-casts the low-precision data type (e.g., INT8) to a wider type (e.g., INT16) for transformed matrices (marked as ❶), as shown in Figure 2(a). While there will be no overflow when using the up-casting approach, the multiplication operations have to be computed under the wider low-precision data type, degrading the desired acceleration that benefits from efficient low-precision computations.
- **Down-Scaling Approach.** Another prevailing solution is the down-scaling approach, which down-scales the transformed matrix by multiplying a scaling factor followed by a rounding operation, as shown in Figure 2(b). For example, we have $\alpha = \frac{1}{4}$ according to the input transformation matrix in Eq. 2, and thus the transformed input can be represented as $round(\frac{1}{4} \times V)$. As the down-scaled values suffer from round-off errors, this approach introduces an extra precision loss for quantized neural networks (marked as ❷). Moreover, 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 [23]). Therefore, the down-scaling approach can hardly be applied to low-precision Winograd convolutions with large m values due to the non-negligible precision loss incurred by excessively down-scaling and rounding operations.

Existing implementations suffer from the disadvantages in performance degradation or precision loss, motivating our work which targets a more effective approach for implementing low-precision Winograd convolutions. Although there is existing work applying

even lower precision integers (e.g., INT4) to quantize the deep learning models [4, 5, 43, 45], it is worth noting that they are hard to apply to the Winograd algorithm due to its numerical instability.

3 QUANTIZATION METHODOLOGY

In this paper, we propose LoWino, a novel approach to effectively combine the Winograd algorithm with low-precision computations, thereby exploiting the INT8 arithmetic on modern architectures. We perform quantization in the Winograd domain after the value range is amplified to avoid the numerical overflow in transformation computations as described in Section 2.2. After applying quantization and de-quantization operations to Eq. 1, our low-precision Winograd convolution is represented as:

$$\boldsymbol{y}_k = A^{\mathrm{T}}\left(Q'\left(\sum_{c=1}^{C} Q\left(G\boldsymbol{g}_{k,c}G^{\mathrm{T}}\right)\tilde{\odot}Q\left(B^{\mathrm{T}}\boldsymbol{d}_c B\right)\right)\right)A \quad (3)$$

where $Q(x)$ and $Q'(x)$ are quantization and de-quantization functions, and $\tilde{\odot}$ denotes the low-precision element-wise multiplication. We employ a linear quantization function with saturation to approximately represent 32-bit floating-point (FP32) values by low-precision 8-bit values:

$$Q(X_{\mathtt{FP32}}) = (\mathcal{S}_{\mathtt{INT8}}(\alpha X_{\mathtt{FP32}}))_{\mathtt{INT8}} \quad (4)$$

In Eq. 4, $\mathcal{S}_{\mathtt{INT8}}$ is a conversion function with saturation, converting FP32 values to integers and limiting the converted values to a fixed range between the minimum and maximum value of INT8, i.e., if a value is greater (less) than the maximum (minimum), it is set to the maximum (minimum). The scaling factor α, which is used to quantize values from the original data type (FP32) to the low-precision data type (INT8), is calculated by:

$$\alpha = (2^{b-1}-1)/\tau \quad (5)$$

where b is the bit-width of low-precision data type, and τ is the threshold to represent the quantization range of full-precision values, i.e., $(-|\tau| \sim +|\tau|)$. Correspondingly, the de-quantization function that recovers the low-precision values to full-precision ones can be represented as:

$$Q'(X_{\mathtt{INT8}}) = (\alpha^{-1}X_{\mathtt{INT8}})_{\mathtt{FP32}} \quad (6)$$

While the parameter τ can be directly set to $||X_{\mathtt{FP32}}||_\infty$, it is usually not the optimal. In practice, we select a reasonable τ by utilizing a calibration process [29] on a small number (~500s) of unlabeled sample images, which is represented as:

$$\tau = \underset{\tau'}{arg\,min}\left(D_{KL}\left(P\left(X_{\mathtt{FP32}}\right)||P\left(Q_{\tau'}\left(X_{\mathtt{FP32}}\right)\right)\right)\right) \quad (7)$$

$X_{\mathtt{FP32}}$ and $Q(X_{\mathtt{FP32}})$ are the full-precision data and corresponding quantized data, $P(X)$ is the discrete probability distribution of X, and D_{KL} is the Kullback-Leibler divergence [21] between two probability distributions. The τ' represents the threshold used in the quantization function $Q(x)$ (refer to Eq. 5). 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 IMPLEMENTATION AND OPTIMIZATIONS

In this section, we describe the details of our low-precision Winograd convolution implementation, which applies the method described in Section 3. Different from the existing solution [13], which caches intermediate data and only supports one small tile size, our design writes all the intermediate data into the main memory so as to enable large matrix multiplications to be performed, which is more computation efficient. However, combining the design with the methodology described in the previous section leads to the following challenges on the way to an efficient low-precision Winograd convolution solution: *a)* extra memory operations containing non-consecutive memory access, e.g., scattering and gathering operations; *b)* quantization and de-quantization overheads; *c)* data type and layout requirements for low-precision computation instructions; *d)* tall and skinny matrix multiplications.

To tackle the challenges, we apply a series of optimizations in different parts of our implementation and combine them together to work as a whole solution. LoWino consists of input and filter transformation, matrix multiplication, and output transformation, as depicted in Figure 3. We categorize our implementation into transformations (input, filter, and output transformation), which are memory-bound, and matrix multiplication, which is computation-bound. We use the following guidelines to perform optimizations: *1)* Overlapping computation and memory operations as much as possible. *2)* Reducing memory access latency. *3)* Increasing the computation efficiency through vectorization and data reuse.

4.1 Data Layout

Data layout determines how to store data in memory and is very critical to performance. To fully utilize the low-precision computational instructions, we use a customized data layout, inspired by the state-of-the-art full-precision implementations [17, 47]. Our data layout is guided by the following principles: *1)* It is compatible with the low-precision computation instructions (refer to Section 2.1), which is the key difference with the full-precision design; *2)* It supports aligned vector loads and stores, which is essential to write fully vectorized codes; *3)* It is cache-friendly, which restricts the computations to access a small range of memory so as to reduce the cache and TLB misses.

Table 1: Customized Data Layout.

Variable	Data Layout
Input images	$B \times [C/\varphi/\sigma] \times H \times W \times \varphi \times \sigma$
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	$B \times [K/\varphi/\sigma] \times N \times T \times \varphi \times \sigma$
Output images	$B \times [K/\varphi/\sigma] \times H' \times W' \times \varphi \times \sigma$

Table 1 describes our customized data layout. Symbols B, C, K, H, W, H', and W' represents the batch size, input channels, output channels, input image height, input image width, output image height, and output image width, respectively. The filter size is $r \times r$. We use σ to represent the vector length of 32-bit floating-point number, which is 16 for VNNI instruction set, and use φ to represent the number of 8-bit elements in a 32-bit word, which is 4.

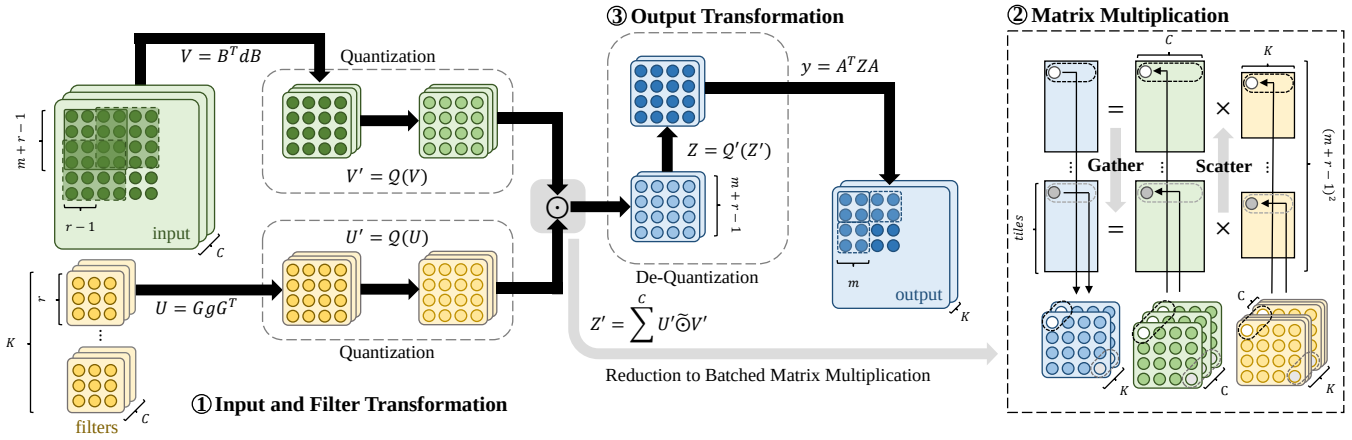


Figure 3: Overview of LoWino using $F(2 \times 2, 3 \times 3)$ as an example. The boxes colored in green, yellow, and blue represent the input tiles, filters, and output tiles, respectively. The dashed rounded-rectangle boxes represent the processes of quantization and de-quantzation.

The input image will be decoupled into N input tiles with overlaps. There are T elements in each single input tile and T is also the number of matrix multiplications to be performed in the Winograd algorithm. C_{blk}, K_{blk}, and N_{blk} are blocking hyper-parameters used in matrix multiplication, which will be elaborated in Section 4.3. As our data layout is customized to cooperate with the computational pattern of low-precision instructions, the matrix multiplication of Winograd convolution can be effectively performed by leveraging VNNI instructions. The adjacent computation operations access a relatively small range of memory, which reduces cache and TLB misses. Furthermore, all data is 64-byte aligned and thus the aligned vectorized load/store instruction can be used.

4.2 Transformation

There are three kinds of transformations in Winograd convolution: input transformation and filter transformation (① in Figure 3), which are performed before matrix multiplications and generate the inputs for it, and output transformation (③ in Figure 3), which takes the matrix multiplication result and converts it back to the spatial domain. All these three transformations are performed between a tile of data and the corresponding transformation matrix (B, G, and A), which are memory-bound and the execution time is dominated by data movements. Thus, we combine the compensation, quantization, and de-quantization with them so as to overlap computation with memory operations, which introduce only trivial overheads while significantly alleviating the burden of matrix multiplication. The following subsections describe the implementations and optimization in each transformation stage.

4.2.1 Input transformation. For each tile d containing 32-bit full-precision numbers, we transform it from the spatial domain to the Winograd domain and then quantize the floating-point numbers to 8-bit integers through the function $Q(x)$ in Eq. 4. The core instruction in matrix multiplication, `vpdpbusd`, requires the first operand (i.e., transformed input) to be unsigned as described in Section 2.1. However, the transformed input can not be guaranteed to be non-negative. Thus, we add 128 to the transformed input after quantization, which is a compensation operation, so as to guarantee all the data can be represented by UINT8. We will perform subtraction operations to recover the result in the matrix multiplication stage, which eliminates the effect introduced by adding 128.

At the end of this stage, we need to store the result in memory for matrix multiplication (② in Figure 3). In general, each result tile needs to be scattered to $T = (m + r - 1)^2$ matrices, resulting in non-consecutive memory writes. To overcome the disadvantage of the operations, we employ two optimizations here. First, based on the customized data layout, each time we store 512-bit data (a whole cache line) into memory to eliminate false sharing. Second, we apply non-temporal stores, which write data in memory directly without fetching data to cache first. The transformed data will not be used in the near future and thus we do not need to worry about the non-temporal stores destroy the cache locality.

4.2.2 Filter Transformation. Filter transformation operates on pre-trained filters. For inference, the filters are known ahead of time so that all the operations in this stage are performed offline and the execution time will not be counted as part of the running time. After quantization, we multiply the result by an auxiliary matrix filled by -128, which is required by the compensation operation so as to guarantee the result's correctness. The final results of filter transformation are reorganized to a specific data layout, where the lowest two dimensions are $(C_{blk}/\varphi) \times (K_{blk} \times \varphi)$ so as to be compatible with the `vpdpbusd` instruction.

4.2.3 Output Transformation. In general, Winograd algorithm [23] triggers gathering operations, which fetch data from $T = (m+r-1)^2$ matrices generated by the matrix multiplication stage, and then performs transformations. However, the data access is not consecutive and computation has to wait till the data is fetched to the cache when using gathering operations. Thus, we instead employ scattering operations at the end of the previous stage (i.e., matrix multiplication) with non-temporal stores, which will be explained in Section 4.3, so that all the data access is consecutive in this stage.

Each time we take a consecutive 32-bit integer data tile generated by matrix multiplication and perform de-quantization, i.e., function $Q'(x)$, on it. To de-quantize the value, we multiply the data tile with the reciprocal of scaling factor after casting 32-bit integers to floating-point values. Then, the transformation procedure is done by multiplying the corresponding transformation matrix.

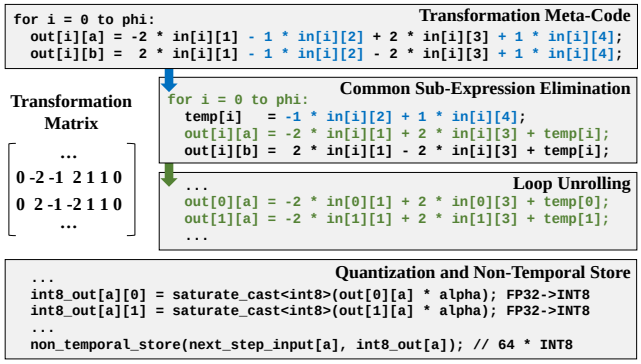


Figure 4: An example of codelets generation.

4.2.4 Codelets. We write a codelet generator for Winograd transformations, which automatically generates C++ codes for a given $F(m, r)$ according to the transformation matrices provided by wincnn [22]. We invoke Intel Intrinsics [11] to guarantee that all the codes are vectorized except for the reorganizing operations in offline filter transformations. Figure 4 illustrates an example of codelet generation for input transformation, which operates $(m+r-1)\times\varphi\times\sigma$ numbers in a column-wise or row-wise computation. The codelets for filter and output transformation are generated similarly. We perform optimizations such as loop unrolling and constant folding on generated codelets. As the transformation matrices have many zeros, we eliminate the redundant operation multiplying an operand by zero. We compute the common terms among different rows in the transformation matrix and store them into intermediate variables, replacing the common terms in the expressions with the intermediate variables, thereby reducing the computational complexity. Furthermore, we unroll the loop of `phi` (φ) to reduce the branches of instructions in codelets, which improves the instruction-level parallelism. By performing in a column-wise manner and then in a row-wise manner on input tiles, the generated codelets are reused to calculate all the transformed inputs.

4.3 Batched Matrix Multiplication

The matrix multiplication (② in Figure 3) dominates the computation of Winograd convolution. The element-wise multiplication of a quantized transformed filter U' and the quantized transformed input tile V' in the same channel is computed under low-precision arithmetic, and subsequently, the results of the multiplication are accumulated along channels (Eq. 3). Therefore, this procedure can be reduced to batched matrix multiplication. We perform a batch size of $T = (m + r - 1)^2$ matrix multiplications, each of which is represented by $Z = V \times U$. The matrix V is generated by the input image transformation with the size of $N \times C$ and the matrix U is generated by the filter transformation with the size of $C \times K$. In general, a convolutional layer consists of a very large number of input tiles (N) and the numbers of input channels (C) and output channels (K) are relatively smaller, causing that the matrix multiplication is performed on tall and skinny matrices. Since tall and skinny matrix multiplication is not highly optimized in off-the-shelf libraries [17, 39], we implement our own batched matrix multiplications for those tall and skinny matrices with the following optimizations: blocking strategies to enhance the data reuse for both cache memory and registers; non-temporal stores to reduce the scattering operation latency; data prefetching to overlap computation with memory operations. The following subsections elaborate those optimizations.

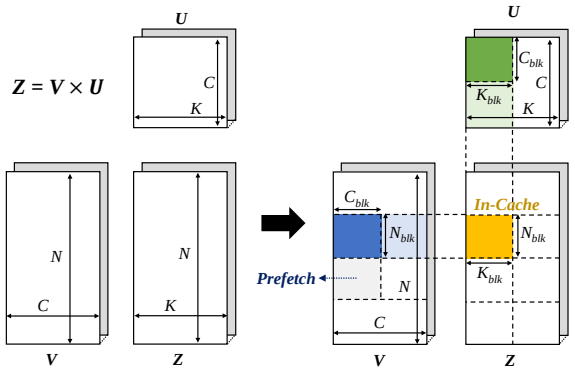


Figure 5: Cache-level matrix blocking strategy. The blocks in blue, green, and yellow represent the blocked sub-matrix of V, U and Z, respectively.

4.3.1 Cache Blocking. In general, it is impossible to guarantee the matrices V, U, and results can fit in the cache. To increase the data reuse, we break the matrices into sub-matrices, and each sub-matrix can fit in L2 cache. The major idea is to fully use the data before swap it out from the cache. Figure 5 depicts our cache blocking method. The matrix V is divided into $\lceil N/N_{blk}\rceil\times\lceil C/C_{blk}\rceil$ sub-matrices of size $N_{blk} \times C_{blk}$, and the matrix U is divided into $\lceil C/C_{blk}\rceil \times \lceil K/K_{blk}\rceil$ sub-matrices of size $C_{blk} \times K_{blk}$. Therefore, the result matrix Z consequently consists of $\lceil N/N_{blk}\rceil \times \lceil K/K_{blk}\rceil$ sub-matrices of size $N_{blk} \times K_{blk}$. Accordingly, each sub-matrix in Z can be accumulated by one row of blocks in V and one column of blocks in U:

$$z_{i,j} = \sum_{k=1}^{C/C_{blk}} v_{i,k} \times u_{k,j} \tag{8}$$

There are C/C_{blk} matrix multiplication results to be accumulated to the sub-matrix z. We store the intermediate accumulation result in a $C_{blk}\times K_{blk}$ buffer, which stays in L2 cache until all the computations in Eq. 8 are completed. After blocking, the matrix u of size $C_{blk} \times K_{blk}$ can be held in L2 cache during the multiplication process in most cases. Besides, we insert prefetching instructions to load $v_{i+1,k}$ into L2 cache when computing $v_{i,k}$. Therefore, when we calculating the next matrix multiplication, i.e., $z_{i+1,j}$, the data is already in-cache, which significantly hides the memory access latency.

4.3.2 Register Blocking. We also implement a register-level blocking strategy to further optimize the multiplication operations of sub-matrices as shown in Figure 6. The idea is similar to cache-level blocking, which divides original matrices into smaller sub-matrices. We reuse the intermediate results by storing them into $row_{blk} \times col_{blk}$ registers, and the final results are scattered into the appropriate locations of tiles used in output transformation by using non-temporal stores. Each row of v is expected to be automatically prefetched into L1 cache by hardware since the memory access is consecutive, and each column of u can also benefit from the hardware prefetcher since we store the data in a column-major format. To follow the low-precision computational instruction, i.e., vpdpbusd, convention, a sub-matrix u is stored in a specific layout, which has been reordered to the size of $(C_{blk}/4) \times (K_{blk} \times 4)$.

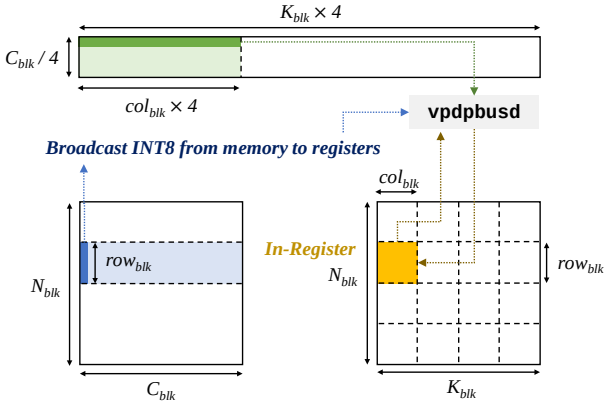


Figure 6: Register-level blocking. The blocks in blue, green, and yellow represent the blocked sub-matrix of v, u and z, respectively.

4.3.3 Compensation. As the vpdpbusd instruction needs the first operand to be unsigned 8-bit integers. We perform compensation operations for this requirement in the input transformation stage (adding 128) and the filter transformation stage (multiplying −128). Thus, the exact computations we perform in this stage are equivalent to the following equation:

$$Z = \hat{V} \times U + \hat{Z} \quad \text{where} \quad \begin{cases} \hat{V} = V + \Delta \\ \hat{Z} = -\Delta \times U \end{cases} \tag{9}$$

The Δ is an auxiliary matrix filled by 128 which has the same size as V. We calculate $\hat{V}$ in the input transformation stage and $\hat{Z}$ in the filter transformation stage. The compensation operation is performed in transformation stages in practice as described in Sections 4.2.1 and 4.2.2. Since the transformation stages have a low compute-to-memory ratio and mostly memory-bound, the extra computation has little effect on the performance.

4.3.4 Code Generation and Tuning. As the configurations of convolutional layers are known at the compile-time, we can utilize the constants, such as the number of loops and the offset of memory access, to optimize the codes. We employ the JIT (just-in-time) compilation technique to generate the codes for matrix multiplication. The pseudo-code for the code template, which computes $z = v \times u + z$, is described in Figure 7.

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

Figure 7: The pseudo-code for matrix multiplication.

The loops of `r0` and `c0` represent the cache-level blocking while `r1` and `c1` represent the register-level blocking. For each loop `r1`, we broadcast a packed 32-bit word containing four 8-bit integers from the source location of v to a 512-bit vector register `v_reg`. For each innermost loop `c1`, we load 64 8-bit integers from u into register `u_reg`, performing vpdpbusd and storing the results into register `z_reg`. The register `v_reg` is reused during this loop. At the end of this stage, we scatter the results to appropriate locations for the next stage usage at the granularity of the cache line by using non-temporal stores. This allows the next stage to access consecutive memory, avoiding expensive gathering operations. To further improve the performance, we fully unroll the loops of `t`, `r1`, and `c1` into consecutive instructions. Then, we reorder the unrolled instructions to mix the software prefetch and computation instructions, thereby improving the instruction-level parallelism. For a specific matrix multiplication operation, the code is generated and compiled as a shared library to compute all sub-matrices.

There are several tuneable parameters for code generation, including N_{blk}, C_{blk}, K_{blk}, row_{blk}, and col_{blk}. We leverage an auto-tuning process to find the optimal parameters. The auto-tuning process takes a relatively small amount of time and usually is performed ahead of time since the convolutional layer's configuration is already known. The optimal parameters are saved into a wisdom file and used in inference. To reduce the search space, we limit the $row_{blk} \times col_{blk} + col_{blk} < 31$ (one extra auxiliary register is used for broadcasting operations in our design), as there are 32 available 512-bit vector registers on modern x86 platforms. We further limit $C_{blk} \times K_{blk}$ to be less than 512^2 so as to ensure the sub-matrices, z, u, and v, can fit in the cache.

4.4 Parallelization

In order to perform the above-mentioned implementation of Winograd convolutions on modern multi-core platforms, we employ a static scheduling strategy to partition the job so as to execute it parallel. As the configurations of convolutional layers are constant during neural network inference, we pre-assign the tasks for all threads at compile-time to reduce run-time overheads. To achieve optimal performance, each thread is assigned the same amount of

computation with the same memory access patterns. We assume there are ω threads, which can be executed simultaneously on multi-core platforms. In the input and output transformation stages, there are N tiles to be operated and each tile contains $T \times \varphi \times \sigma$ elements. Therefore, each thread operates up to $\lceil N/\omega \rceil$ tile transformation tasks. Moreover, we recursively divide the task dimensions so that the tiles to be operated are contiguous for each thread, which can potentially facilitate cache reuses. In the filter transformation stage, each thread operate up to $\lceil (C \times K/\varphi/\sigma)/\omega \rceil$ filter transformation tasks, which is similar as the input transformation stage. In the matrix multiplication stage, there are $N/N_{blk} \times K/K_{blk} \times T$ sub-matrices to be performed and each sub-matrix contains $N_{blk} \times K_{blk}$ elements, according to our blocking strategies (in Section 4.3). Thus, each thread is assigned up to $\lceil (N/N_{blk} \times K/K_{blk} \times T)/\omega \rceil$ matrix multiplication tasks. The job consists of parallel tasks is executed using a single fork-join method. As the C and K, as well as the number of threads ω, are typically powers of two, the tasks can be equally assigned to all the threads. In practice, all threads could start and finish the assigned workload roughly at the same time, resulting in a balanced situation.

5 EVALUATION

In this section, we demonstrate that LoWino is an effective approach for accelerating Winograd convolutions by utilizing low-precision computations while maintaining the accuracy at a reasonable level. Specifically, we address two major research questions:
1) What is the performance of LoWino compared with the state-of-the-art implementations?
2) What is the end-to-end accuracy loss of our approach on representative convolutional neural networks?

Experimental Setup. Our experiments are performed on an 8-core Intel Xeon Scalable Processor @ 3.00GHz (Cascade Lake) platform running a Linux-based operating system, Ubuntu20.04 LTS, which has 32GB global main memory. The programs are compiled by g++ (version 9.3.0). 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 [20], VGG16 [36], ResNet-50 [9], GoogLeNet [37], YOLOv3 [34], FusionNet [33], and U-Net [35]. Following the convention [17], we set the batch size for classification models to 64 and the batch size for object detection and semantic segmentation models to one. 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 [13], including direct and Winograd convolution, which include the quantize and de-quantize steps.

Figure 8 shows the normalized execution time for all the convolutional layers in Table 2. We also report the speedups of our LoWino $F(4 \times 4, 3 \times 3)$ over oneDNN's low-precision Winograd convolutions. Overall, LoWino $F(4 \times 4, 3 \times 3)$ achieves an up to 2.04× speedup and an average of 1.26× speedup over the best implementations in oneDNN. There are three observations. First, LoWino $F(2\times2, 3\times3)$ achieves competitive performance compared with the implementations in vendor library, i.e., oneDNN's 8-bit Winograd convolution. Second, LoWino $F(4 \times 4, 3 \times 3)$ usually is the best performer, delivering significant performance improvement over $F(2 \times 2, 3 \times 3)$. The speedups of LoWino are derived from both the theoretical complexity reduction of the Winograd algorithm and our optimization techniques. Third, Winograd convolution not always outperforms 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. For example, the result of ResNet-50_a shows that the low-precision Winograd convolution with $F(2\times2, 3\times3)$ is slower than direct convolution in both oneDNN's and our implementations. Our $F(4\times4, 3\times3)$ can provide an efficient implementation to run this layer faster. For some special layers, like Yolov3_a, direct convolution outperforms $F(4\times4, 3\times3)$, since the transformation overhead is much larger than the computation savings. Nevertheless, LoWino with $F(4 \times 4, 3 \times 3)$ accelerates the convolution layers for most cases. In addition, compared to the best full-precision implementation in oneDNN, LoWino with $F(2 \times 2, 3 \times 3)$ and $F(4 \times 4, 3 \times 3)$ achieves an average of 1.9× and 2.6× speedups, which validates the performance improvement of low-precision computing.

Table 2: Benchmarked Convolutional Layers.

Layer	B	C	K	H&W	r
AlexNet_a	64	384	384	13	3
AlexNet_b	64	384	256	13	3
VGG16_a	64	256	256	58	3
VGG16_b	64	512	512	30	3
VGG16_c	64	512	512	16	3
ResNet-50_a	64	128	128	28	3
ResNet-50_b	64	256	256	14	3
ResNet-50_c	64	512	512	7	3
GoogLeNet_a	64	128	192	28	3
GoogLeNet_b	64	128	256	14	3
GoogLeNet_c	64	192	384	7	3
YOLOv3_a	1	64	128	64	3
YOLOv3_b	1	128	256	32	3
YOLOv3_c	1	256	512	16	3
FusionNet_a	1	128	128	320	3
FusionNet_b	1	256	256	160	3
FusionNet_c	1	512	512	80	3
U-Net_a	1	128	128	282	3
U-Net_b	1	256	256	138	3
U-Net_c	1	512	512	66	3

5.2 Neural Network Accuracy

We choose VGG16 [36] and ResNet-50 [9] as two representative convolutional neural networks to evaluate the end-to-end model accuracy of our approach on the ImageNet dataset [6]. 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 [14, 19, 29, 31, 44] which leverage normal (non-Winograd) low-precision convolutions. As $F(4\times4, 3\times3)$ is not supported in oneDNN, we implement the down-scaling

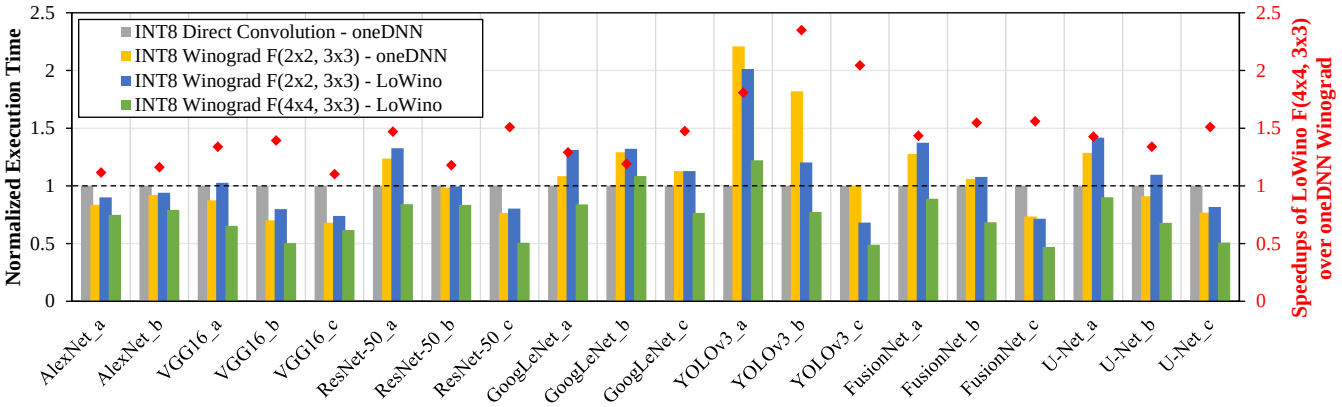


Figure 8: Normalized execution time for different convolution layers.

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 [32] as the baseline. Since the different hyper-parameters might be 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 [14, 19, 29, 31, 44], the accuracy numbers are directly cited from the corresponding papers.

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	Non-Winograd Covolution	KLD [29]	69.40	69.20
		Yao *et al.* [44]	69.40	69.10
	F(2×2, 3×3)	oneDNN [13]	71.59	70.98
		LoWino (Ours)	71.59	71.33
	F(4×4, 3×3)	Down-Scaling Impl.	71.59	00.00
		LoWino (Ours)	71.59	69.20
ResNet-50	Non-Winograd Convolution	KLD [29]	73.23	73.10
		Yao *et al.* [44]	75.30	74.80
		Jacob *et al.* [14]	76.40	74.90
		Park *et al.* [31]	77.72	75.67
		Krishnamoorthi [19]	75.20	75.10
	F(2×2, 3×3)	oneDNN [13]	76.13	75.91
		LoWino (Ours)	76.13	76.09
	F(4×4, 3×3)	Down-Scaling Impl.	76.13	00.00
		LoWino (Ours)	76.13	75.53

The accuracy of quantized neural networks that utilize low-precision Winograd convolutions is similar to those non-Winograd convolutions. Benefited from our quantization design, LoWino achieves less accuracy loss compared with the down-scaling approach when using $F(2 \times 2, 3 \times 3)$. As discussed in Section 2.3, the scaling factor of $F(4 \times 4, 3 \times 3)$ is much less than the factor of $F(2 \times 2, 3 \times 3)$, leading to much more precision loss incurred by down-scaling and rounding operations. The experimental result shows that the down-scaling approach with $F(4 \times 4, 3 \times 3)$ 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 $F(2 \times 2, 3 \times 3)$ and $F(4 \times 4, 3 \times 3)$. These results demonstrate the effectiveness of LoWino, which can accelerate convolutional layers while achieving tolerable accuracy loss.

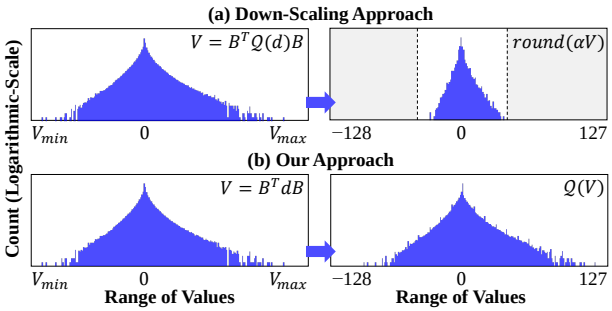


Figure 9: Comparing the down-scaling approach with ours for $F(4 \times 4, 3 \times 3)$ low-precision Winograd convolution.

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 9 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 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 numbers 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 are quantized in the Winograd domain. Thus, we can fully use the numbers of [-128...127] to represent the original values, thereby reducing the precision loss.

5.3 Execution Time Breakdown

In Figure 8, we observe that for some layers, LoWino $F(2\times2, 3\times3)$ is better than oneDNN's Winograd convolution counterpart, and for some layers, oneDNN's is better. We select VGG16_b, ResNet-50_c, YOLOv3_c, and U-Net_b as examples to perform a deeper analysis of this phenomenon. In Figure 10, we use oneDNN's implementation as baselines and show the normalized execution time. We divide the execution time into matrix multiplication, which is compute-bound, and transformation, which is memory-bound. We observe that our approach owns more transformation time than oneDNN's. This is because that our input data type is 32-bit floating-point, whereas oneDNN uses 8-bit integers. Since the transformation process is memory-bound, our approach reads 4× input data compared to oneDNN, resulting in more execution time. However, loading 32-bit floating-point numbers is necessary to guarantee the accuracy as explained in Section 2.3.

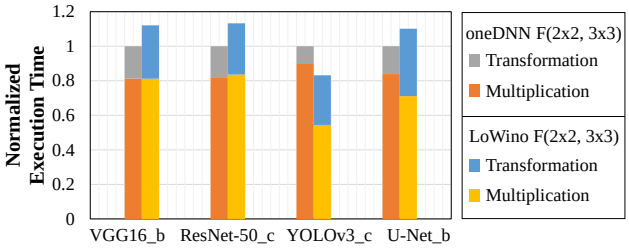


Figure 10: Analyzing the performance of different parts in low-precision Winograd convolutions.

Due to different design decisions, LoWino and oneDNN have different matrix multiplication performances. The oneDNN divides the input data into several partitions, and for each part, it saves all the intermediate data. While this implementation decreases the data loading and storing time, the matrix multiplication has to be performed between two small matrices due to the limited cache size, resulting in a low compute-to-memory ratio. Since the intermediate data size is increased with the tile size, the input partition size has to be reduced so as to fit in the cache, which results in smaller matrix multiplications and less computational efficiency. For instance, $F(4\times4, 3\times3)$ has 2.25× intermediate comparing with $F(2\times2, 3\times3)$ for a single tile. Whereas, our approach writes all intermediate data into the main memory and reads them in the matrix multiplication stage. As we do not cache all the intermediate data, larger block sizes can be used comparing with oneDNN. For the layer which does not have large enough matrices, our blocking size is similar to oneDNN's. Thus, the execution time of matrix multiplication for VGG16_b and ResNet-50_c is also similar. However, once the layer has larger matrices, our approach uses larger block sizes, resulting in a higher compute-to-memory ratio and higher performance. That's why for YOLOv3_c and U-Net_b, the performance of our matrix multiplication is better than that of oneDNN. For Winograd convolutions, the matrix size is decided by the number of input channels and output channels. Therefore, for layers with large input and output sizes, LoWino usually performs better.

6 RELATED WORK

To reduce the complexity of the compute-intensive convolutions in CNNs, Lavin and Gray [23] introduced the approach applying Winograd's minimal filtering algorithm on convolutional layers and demonstrated its effectiveness. Subsequently, the Winograd-based convolution was integrated into popular vendor libraries and be widely used to accelerate convolutional neural networks. In recent years, several efforts have been conducted on optimization techniques for efficient Winograd convolutions on CPU [17, 18, 46] and heterogeneous platforms [16, 28, 42]. Jia *et al.* [17] proposed an implementation to support n-dimensional Winograd-based convolutions on many-core CPUs. Mazaheri *et al.* [28] optimized Winograd convolution via symbolic computation and meta-programming. Yan *et al.* [42] built an assembler to perform assembly-level optimizations. Besides, Huang *et al.* [10] presented a decomposable Winograd method which supports convolution layers with large kernels and large strides.

The low-precision quantization techniques [14, 19, 29, 31, 44], which convert full-precision models into low-precision ones to exploit integer-arithmetic computations, are expected to further improve the performance of Winograd convolutions [25, 40]. The ncnn [38] has low-precision Winograd convolution implementation using an up-casting approach, while oneDNN [13] provides a down-scaling approach solution. However, these approaches in vendor libraries suffer from the disadvantages of accuracy loss or performance degradation, as revealed in Section 2.3. Besides, some studies attempt to leverage re-training and searching on the whole training dataset [7, 24]. Those methods rely on re-training to reduce the accuracy loss introduced by quantization and Winograd convolutions, whereas our approach reduces the accuracy loss through performing the 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.

7 CONCLUSION

In this paper, we proposed a low-precision Winograd convolution approach, namely LoWino, which introduces quantization in the transformed domain, thereby effectively exploiting the capability of low-precision computations while maintaining the accuracy at a reasonable level. Experimental results show that our approach delivers an up to 2.04× speedup and an average of 1.26× speedup over the best implementations in the existing vendor library. In future work, we plan to: 1) explore an automatic mechanism to select the optimal algorithm for a convolutional layer among direct, Winograd, and others; 2) incorporate our approach to a deep learning compiler that supports low-precision computation (e.g., [15]) and apply it to more modern architectures.

ACKNOWLEDGMENTS

The authors thank the anonymous reviewers of the paper for their valuable comments. Guangli Li and Xiaobing Feng are grateful for

the funding support from the National Key R&D Program of China under Grant No.2017YFB1003103, and the Science Fund for Creative Research Groups of the National Natural Science Foundation of China under Grant No.61521092.

REFERENCES

[1] 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.

[2] Yunji Chen, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. 2016. DianNao family: energy-efficient hardware accelerators for machine learning. *Commun. ACM* 59, 11 (2016), 105–112.

[3] 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.

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

[5] 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).

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

[7] 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.

[8] 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).

[9] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 770–778.

[10] 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 *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 34. 4174–4181.

[11] Intel. 2021. *Intrinsics Guide*. Retrieved March 29, 2021 from https://software.intel.com/sites/landingpage/IntrinsicsGuide/

[12] 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

[13] Intel. 2021. *oneAPI Deep Neural Network Library (oneDNN)*. Retrieved February 27, 2021 from https://github.com/oneapi-src/oneDNN

[14] 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 *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2704–2713.

[15] Animesh Jain, Shoubhik Bhattacharya, Masahiro Masuda, Vin Sharma, and Yida Wang. 2020. Efficient Execution of Quantized Deep Learning Models: A Compiler Approach. (2020). arXiv:2006.10226 [cs.DC]

[16] 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.

[17] Zhen Jia, Aleksandar Zlateski, Fredo Durand, and Kai Li. 2018. Optimizing N-dimensional, winograd-based convolution for manycore CPUs. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 109–123.

[18] Zhen Jia, Aleksandar Zlateski, Fredo Durand, and Kai Li. 2018. Towards Optimal Winograd Convolution on Manycores.

[19] Raghuraman Krishnamoorthi. 2018. Quantizing deep convolutional networks for efficient inference: A whitepaper. *arXiv preprint arXiv:1806.08342* (2018).

[20] 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.

[21] Solomon Kullback. 1997. *Information theory and statistics*. Courier Corporation.

[22] Andrew Lavin. 2021. *wincnn*. Retrieved February 27, 2021 from https://github.com/andravin/wincnn

[23] Andrew Lavin and Scott Gray. 2016. Fast algorithms for convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 4013–4021.

[24] 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 *2020 IEEE International Conference on Acoustics, Speech and Signal Processing*. IEEE, 3842–3846.

[25] 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 *2021 IEEE/ACM International Symposium on Code Generation and Optimization*. IEEE, 90–102.

[26] 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.

[27] Michael Mathieu, Mikael Henaff, and Yann LeCun. 2014. Fast training of convolutional networks through FFTS: International Conference on Learning Representations. In *2nd International Conference on Learning Representations*.

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

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

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

[31] 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.

[32] 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.

[33] Tran Minh Quan, David GC Hildebrand, and Won-Ki Jeong. 2016. Fusionnet: A deep fully residual convolutional neural network for image segmentation in connectomics. *arXiv preprint arXiv:1612.05360* (2016).

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

[35] 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.

[36] Karen Simonyan and Andrew Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *3rd International Conference on Learning Representations*.

[37] 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 *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 1–9.

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

[39] 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 *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–12.

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

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

[42] Da Yan, Wei Wang, and Xiaowen Chu. 2020. Optimizing batched winograd convolution on GPUs. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 32–44.

[43] Haojin Yang, Martin Fritzsche, Christian Bartz, and Christoph Meinel. 2017. Bmxnet: An open-source binary neural network implementation based on mxnet. In *Proceedings of the 25th ACM international conference on Multimedia*. 1209–1212.

[44] 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.

[45] Zhewei Yao, Zhen Dong, Zhangcheng Zheng, Amir Gholami, Jiali Yu, Eric Tan, Leyuan Wang, Qijing Huang, Yida Wang, Michael W Mahoney, et al. 2020. HAWQV3: Dyadic Neural Network Quantization. *arXiv preprint arXiv:2011.10680* (2020).

[46] Aleksandar Zlateski, Zhen Jia, Kai Li, and Fredo Durand. 2018. A Deeper Look at FFT and Winograd Convolutions.

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