Im2win: Memory Efficient Convolution On SIMD Architectures

Shuai Lu
Nanchang Hangkong University
Nanchang, China
20160854001010@stu.nchu.edu.cn

Jun Chu
Nanchang Hangkong University
Nanchang, China
chuj@nchu.edu.cn

Xu T. Liu
University of Washington
Seattle, USA
x0@uw.edu

***Abstract*—Convolution is the most expensive operation among neural network operations, thus its performance is critical to the overall performance of neural networks. Commonly used convolution approaches, including general matrix multiplication (GEMM)-based convolution and direct convolution, rely on im2col for data transformation or do not use data transformation at all, respectively. However, the im2col data transformation can lead to at least 2× memory footprint compared to not using data transformation at all, thus limiting the size of neural network models running on memory-limited systems. Meanwhile, not using data transformation usually performs poorly due to nonconsecutive memory access although it consumes less memory. To solve those problems, we propose a new memory-efficient data transformation algorithm, called im2win. This algorithm refactorizes a row of square or rectangle dot product windows of the input image and flattens unique elements within these windows into a row in the output tensor, which enables consecutive memory access and data reuse, and thus greatly reduces the memory overhead. Furthermore, we propose a high-performance im2win-based convolution algorithm with various optimizations, including vectorization, loop reordering, etc. Our experimental results show that our algorithm reduces the memory overhead by average to 41.6% compared to the PyTorch's convolution implementation based on im2col, and achieves average to 3.6× and 5.3× speedup in performance compared to the im2col-based convolution and not using data transformation, respectively.**

***Index Terms*—Convolution algorithm, im2win, parallel computing, convolutional neural networks.**

I. Introduction

Convolution is one of the most important components within neural network models, not only because it helps filter critical features out of massive data efficiently, but also because it is the most expensive operation compared with other operations [1]. In particular, a recent study [2] reported that, in CNN inference, there are 50%-90% of the total operations, including pooling, ReLU, and fully-connected, that are convolution operations. Also, convolution accounts for over 90% of the total execution time of many popular neural networks [3]. Therefore, it is essential to reduce the cost of convolution operations in order to improve performance of neural networks.

There are mainly two groups of convolution approaches with respect to data transformation within production machine learning frameworks, such as PyTorch [4] and TensorFlow [5]: im2col data transformation-based and not using data transformation at all, also known as direct convolution. Im2col data transformation takes elements of each dot product window, flattens them out and stacks them up row by row in the output matrix, called im2col matrix. Im2col-based convolution operation starts by transforming the input tensor into im2col matrix by im2col algorithm, and the filter is unfolded into a 2D matrix, followed by a GEMM function call using Basic Linear Algebra Subprograms (BLAS) [6] to complete the compute operation, and finally transposes the resultant matrix into a high-dimensional tensor. The problem with the im2col-based convolution is that the im2col operation generates a high memory footprint and bandwidth overhead. As shown in Figure 1, the im2col transformation produces a matrix with many duplicate elements, for example, the elements in the green box are stored twice in the im2col matrix. In the rest of the paper, we assume all data (such as images, matrices, and tensors) are stored as an 1D array in memory. The resultant im2col matrix is typically larger than the input image (36 elements versus 16 elements).

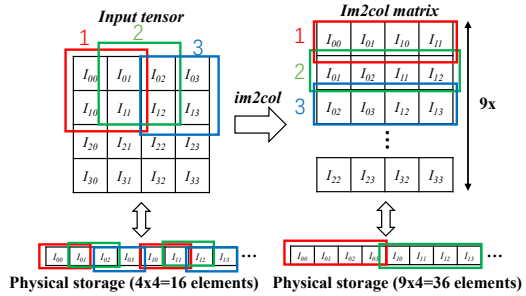


Fig. 1: Illustration of the im2col transformation on a 1×1×4×4 input tensor and a 1×1×2×2 filter tensor. The red, green and blue boxes indicate the window for the first three dot product operations. Note after transformation, im2col matrix has 36 elements.

Convolution approaches without data transformation often use direct convolution algorithm [7]. A typical direct convolution is implemented with seven nested loops over the original input tensor with scalar alpha times x plus y (AXPY) computations. This method has no additional memory overhead compared to im2col-based (GEMM) convolution, however, its memory access is nonconsecutive. As shown on the left in Figure 1, each dot product involving the window data is nonconsecutive memory access, resulting low data reuse and cache hit. The larger the input tensor is, the more

the performance of direct convolution degrades.

To solve those problems, we propose a new memory-efficient data transformation algorithm, called the im2win algorithm. In contrast with im2col, our im2win algorithm refactorizes a row of square or rectangle dot product windows of the input image and flattens unique elements within these windows into a row in the output tensor, which enables consecutive memory access and data reuse, and thus greatly reduces the memory overhead. Note that there may be duplicate elements between row and row. Furthermore, we propose a high-performance im2win-based convolution algorithm with various optimizations, including vectorization and fused-multiply-add (FMA) instructions, loop reordering, hoist and loop unrolling, register and cache blocking, and parallelization strategy. Our experimental results show that our algorithm reduces the memory overhead by average to 41.6% compared to the PyTorch's up-to-date convolution implementation based on im2col, and achieves on average 3.6× and 5.3× speedup in performance compared to the im2col-based convolution and not using data transformation, respectively. We make our code publicly available at https://github.com/ls110082/Im2win.

Contributions. The main contributions of this paper are:

1) We propose an **im2win data transformation algorithm**, which not only greatly reduces the memory footprint for convolutions, but also provides better data locality. This contributes to our overall 41.6% memory reduce against PyTorch's im2col-based convolution.

2) We propose an **im2win-based convolution algorithm** which makes the best use of the data reuse and locality provided by our im2win data transformation algorithm.

3) We propose a collection of optimizations for **the im2win-based convolution algorithm** on SIMD architecture with vectorization instruction set and various optimizations.

II. Preliminaries

In this section, we define the notations used in this paper, review the im2col-based convolution and direct convolution, and the related works.

A. Notations

The three main tensor data in the convolution operation are the Input tensor ($\mathcal{I}$), the Filter tensor ($\mathcal{F}$), and the Output tensor ($\mathcal{O}$). These tensors in NCHW layout are expressed as $\mathcal{I}[N_i][C_i][H_i][W_i]$, $\mathcal{F}[C_o][C_i][H_f][W_f]$ and $\mathcal{O}[N_i][C_o][H_o][W_o]$. The convolution is defined as:

$$\mathcal{O}_{(i,j,m,n)} = \sum_{j=0}^{C_i-1} \sum_{m=0}^{H_f-1} \sum_{n=0}^{W_f-1} \left(\mathcal{I}_{(i,j,m\times s+u,m\times s+v)} \times \mathcal{F}_{(j,r,u,v)}\right), \quad (1)$$

subject to

$$i = 0,1,..,N_i-1, j = 0,1,..,C_o-1, m = 0,1,..,H_o-1,$$
$$n = 0,1,..,W_o-1, u = 0,1,..,H_f-1, v = 0,1,..,W_f-1,$$
$$r = 0,1,..,C_i-1.$$

N_i is the batch size, s is the stride size, C_i and C_o are the number of input and output channels, $H_{i/f/o}$ and $W_{i/f/o}$ denote height and width in spatial dimensions.

B. The im2col-based Convolution

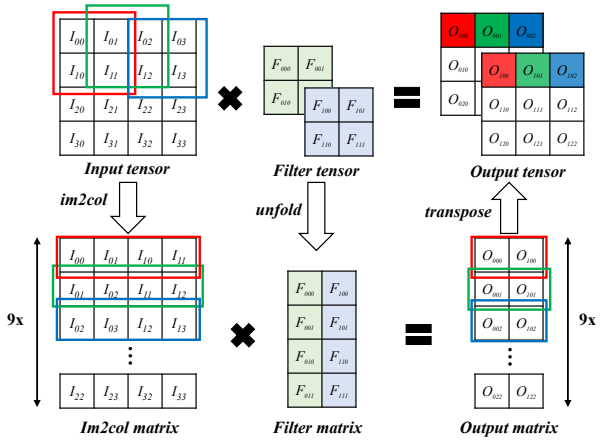


Fig. 2: Basic convolution and im2col+GEMM convolution examples with $H_i = W_i = 4$, $H_f = W_f = 2$, $s_h = s_w = 1$, $H_o = W_o = 3$ ($C_i = C_f = 1$, $C_o = 2$). The different colored boxes denote the correspondence between the input image and the output features.

Proposed by Chellapilla et al. [8], the im2col-based convolution is the simplest and most commonly used convolution algorithm, and it is widely used in existing deep learning frameworks [4], [5], [9]. Due to its fundamental and general nature, it is often used as a benchmark for comparison. The im2col-based convolution unrolls the convolution operation into a general matrix multiplication operation. The $\mathcal{I}$ of size $N_i \times C_i \times H_i \times W_i$ is processed in N_i batches, each batch contains data $\mathcal{I}'$ of size $C_i \times H_i \times W_i$ (i.e., a single image). As shown in Figure 2, the im2col algorithm transforms $\mathcal{I}'$ into a 2D matrix; and $\mathcal{F}$ is unfolded into a filter matrix. In im2col, the elements of each dot product window of $\mathcal{I}'$ is flattened and copied into a single row of a matrix (see Figure 1). Denoting the im2col matrix as M and the filter matrix as N, the im2col algorithm can be written as: $M(mW_o+n, (rH_f+u)W_f+v) = \mathcal{I}'(r, m+u, n+v)$, $N((rH_f+u)W_f+v, j) = \mathcal{F}(j, r, u, v)$. Next, a GEMM operation in BLAS library performs the matrix product of the transformed input matrix and the transformed filter matrix to get the output matrix: $R' = M \times N$. The convolution result tensor R is transposed from R': $R(j, m, n) = R'(mW_o + n, j)$.

Benefiting from the highly optimized BLAS library, the im2col-based convolution has reliable performance and supports various input tensor sizes [8]. However, the performance of this convolution depends heavily on the performance of the GEMM operation. Previous work finds that when the input matrices vary significantly in size and shape, the GEMM operation performs poorly on hierarchical memory architectures [10]. Because the im2col matrix is much larger than the filter matrix, this results the GEMM operation in significantly lower performance than the best achievable performance. Several im2col-based works [11], [12] and MEC [13] split a single

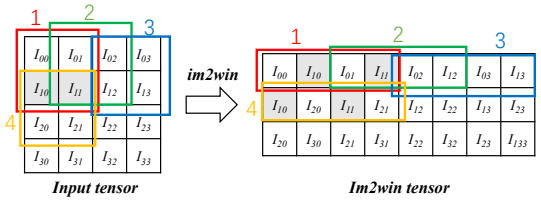


Fig. 3: Illustration of the im2win data transformation from the original tensor to the im2win tensor. The different colored boxes indicate the mapping of elements between the input and the im2win tensors. After transformation, the im2win tensor has 24 elements.

matrix multiplication into multiple small matrix multiplications to reduce the storage overhead. MEC proposes a matrix lowering scheme for the input matrix to reduce the memory footprint, then performs multiple GEMM operations in parallel to complete the convolution. Different from these approaches, we do not perform matrix multiplication in our convolution, instead, we transform the input tensor and perform convolution on the transformed tensor.

C. Direct Convolution

The direct convolution process is described in the top of Figure 2. It performs direct convolution on $\mathcal{I}$ and $\mathcal{F}$ without transformation, hence it has low memory and bandwidth usage compared with the im2col-based convolution. A basic direct convolution has seven nested for loops. The outer four loops iterate over the four dimensions of $\mathcal{O}$, and the inner three loops iterate over $\mathcal{F}$ and $\mathcal{I}$. The result is obtained by performing a basic AXPY operation within the above loops. Direct convolution costs no additional memory but accesses data nonconsecutively therefore has low performance.

Some recent works in high performance computing have focused on direct convolution. Intel LibXSMM library employs parameterized architecture-specific Just-in-Time code generator to optimize the implementation of direct convolution [14]. It has been shown that the performance of direct convolution can be greatly improved by designing specific data layouts based on the loop ordering of the algorithm [7]. The optimizations on SIMD architecture have also been proposed [15]. Different from the above works, our work provides consecutive memory access and reduces additional memory cost by making the best use of the proposed im2win data transformation.

III. Our Im2win-based Convolution

To solve the huge memory footprint of the im2col-based convolution, and the nonconsecutive memory access of the direct convolution, we present our im2win data transformation algorithm and our high-performance im2win-based convolution algorithm. In addition, we will elaborate on the implementation of our high-performance im2win-based convolution algorithm and the optimization techniques.

A. Proposed Im2win Algorithm

We propose an image to window algorithm (called im2win), which dramatically reduce memory overhead by more compact

Algorithm 1: Im2win Algorithm

Input: Input tensor $\mathcal{I}$, Filter tensor $\mathcal{F}$, Stride s
Output: Im2win tensor $\hat{\mathcal{I}}$

1 $H_o = (H_i - H_f)/s + 1$
2 **if** $H_f > s$ **then**
3 Allocate $\hat{\mathcal{I}}$ with $N_i \times H_o \times C_i \times H_f \times W_i$
4 **else**
5 Allocate $\hat{\mathcal{I}}$ with $N_i \times C_i \times H_i \times W_i$
6 **for** $i = 0$ **to** $N_i - 1$ **in parallel do**
7 **for** $r = 0$ **to** $C_i - 1$ **do**
8 **for** $m = 0$ **to** $H_o - 1$ **do**
9 **for** $k = 0$ **to** $W_i - 1$ **do**
10 **for** $u = 0$ **to** $H_f - 1$ **do**
11 $\hat{\mathcal{I}}[i][r][m][k * H_f + u] = \mathcal{I}[i][r][m * s + u][k]$

data arrangement compared with im2col. As shown in Figure 3, our algorithm transforms $\mathcal{I}$ in the order of the dot product window in the convolution operation. For two consecutive dot product window operations, most of the elements loaded in the first operation are reused in the second operation, which can greatly improve data reusability and cache hit.

In the im2win algorithm, we divide each channel of $\mathcal{I}'$ into $H_o \times W_o$ windows of size $H_f \times W_f$, and copy W_o windows in the same row to one row in our im2win tensor. Performing the above operation for all windows on a single channel of $\mathcal{I}'$, we obtain a tensor of size (H_o, $H_f \times W_i$) (see Figure 3). This tensor is ordered by the dot product windows and has fewer redundant elements than what the im2col matrix has. Performing the above algorithm for the batch and channel dimensions in $\mathcal{I}$, we will get a tensor of size (N_i, C_i, H_o, $W_i \times H_f$) and call this tensor as an im2win tensor. Denoting the im2win tensor as $\hat{\mathcal{I}}$, the algorithm can be written as:

$$\hat{\mathcal{I}}(i, r, m, kH_f + u) = \mathcal{I}(i, r, m + u, n + v). \quad (2)$$

subject to

$$m = 0, 1, .., H_o - 1, n = 0, 1, .., W_o - 1, u = 0, 1, .., H_f - 1,$$
$$v = 0, 1, .., W_f - 1, i = 0, 1, .., N_i - 1, r = 0, 1, .., C_i - 1,$$
$$k = 0, 1, .., W_i - 1.$$

The im2win algorithm is described in Algorithm 1. At Line 1, we first calculate the height of $\mathcal{O}$ ($\mathcal{O}$ is computed in the subsequent im2win-based convolution algorithm). Next we decide the size/dimension of the im2win tensor $\hat{\mathcal{I}}$ from Line 3 to Line 5. When $H_f > s$, the lower part of the dot product windows in the previous row and the upper part of the dot product windows in the current row have the same elements, therefore we allocate memory of size $N_i \times H_o \times C_i \times H_f \times W_i$ to $\hat{\mathcal{I}}$ (Line 3). For example in Figure 3 $H_f = 2$ and $s = 1$, therefore $H_f > s$. Note that in the input tensor, the elements in the second row of the first dot product window (the red box) and the elements of the first row of the fourth dot product window (the orange box) are the same (colored in gray). Similarly, the second row of the second window and first row

of the fifth window are the same; the second row of the third window and the first row of the sixth window are the same. When $H_f \leq s$, there are no common elements in the lower part of the dot product windows in the previous row and the upper part of the dot product windows in the current row. In this case, the size of $\hat{\mathcal{I}}$ is the same as that of $\mathcal{I}$. We allocate memory of size $N_i \times C_i \times H_i \times W_i$ to $\hat{\mathcal{I}}$ (Line 5).

Next, the elements of $\mathcal{I}$ are copied into $\hat{\mathcal{I}}$. Recall that the underlying storage of all data is an 1D array in memory. If the tensor is stored in the order of $NCHW$ structure, the distance of the memory addresses of two contiguous elements in the W dimension is one and the distance of the memory addresses of two contiguous elements in the H dimension is W. In hierarchical memory architecture, because elements with consecutive memory addresses tend to be pulled up the memory hierarchy together. The shorter the distance of the memory addresses of the elements is, the better the spatial locality is. Because of this, the access cost for N, C, H, W dimensions decreases respectively. Hence we prioritize the loop order (Line 6 - Line 10) based on the access cost of $\hat{\mathcal{I}}(N_i,\ C_i,\ H_o,\ W_i \times H_f)$ when copying the elements from $\mathcal{I}$ into $\hat{\mathcal{I}}$ (Line 11).

Now we compare the memory usage of the im2win tensor with that of the im2col matrix. Suppose the size of $\mathcal{I}$ is $N_i \times C_i \times H_i \times W_i$, the size of $\mathcal{F}$ is $N_f \times C_f \times H_f \times W_f$, and the stride is s (notice that $H_f \not< s$ and $W_f \not< s$). When $H_f > s$, the size of the im2win tensor is $N_i \times C_i \times H_f \times W_i \times ((H_i - H_f)/s + 1)$. When $H_f = s$, the size of the im2win tensor is equal to the size of $\mathcal{I}$. Additionally, the size of the im2win tensor has no correlation with the value of W_f. When $H_f > s$ or $W_f > s$, the size of the im2col matrix is $C_i \times H_f \times W_f \times ((H_i - H_f)/s + 1) \times ((W_i - W_f)/s + 1)$, which is greater than the size of $\mathcal{I}$. When $H_f = W_f = s$, the size of the im2col matrix is equal to that of $\mathcal{I}$. Assume $H_f = W_f$, the size difference of the im2col matrix and the im2win tensor is $\delta = H_f \times C_i \times (W_i - W_f) \times ((H_i - H_f)/s + 1) \times (W_f/s - 1)$, where $H_o, H_f, C_i, W_i, W_f, s$ are all greater than 0 and $W_i \geq W_f$. When $H_f = W_f = s$, we get $\delta = 0$, i.e., the im2win tensor and the im2col matrix has the same size of $\mathcal{I}$. When $H_f = W_f > s$, we get $\delta > 0$, i.e., the size of the im2col matrix is greater than that of the im2win tensor. In summary, the size of the im2col matrix is greater than that of the im2win tensor no matter what. Recall in Figure 1 $s = 1$, the im2col matrix has 36 elements, while in Figure 3, our im2win tensor has 24 elements. Our im2win tensor has $1/3$ less elements than the im2col matrix in addition to better data locality.

B. Our Im2win-based Convolution Algorithm

We propose an im2win-based convolution, as shown in Algorithm 2. The input tensor $\mathcal{I}$ is transformed into the im2win tensor $\hat{\mathcal{I}}$ at Line 1. Next, the convolution is implemented as multiple nested for loop structure akin to direct convolution. In the innermost loop, the most expensive computation is the AXPY operation at Line 9. Algorithm 2 is the basic implementation before optimization. We propose a composition of optimizations making the best use of the im2win data transformation. These optimizations include vectorization and FMA instructions, loop reordering, hoist and loop unrolling, register and cache blocking, and a parallelization strategy.

Algorithm 2: Basic im2win-based Convolution

Input: Input tensor $\mathcal{I}$, Filter tensor $\mathcal{F}$, Stride s
Output: Output tensor $\mathcal{O}$

```
1 Î[N_i][C_i][H_o][W_i × H_f] = Function IM2WIN(I, F, s)
2 for i = 0 to N_o − 1 do
3     for j = 0 to C_o − 1 do
4         for m = 0 to H_o − 1 do
5             for n = 0 to W_o − 1 do
6                 for r = 0 to C_f − 1 do
7                     for u = 0 to H_f − 1 do
8                         for v = 0 to W_f − 1 do
9                             O[i][j][m][n] +=
                                Î[i][r][m][n ∗ s ∗ W_f + v ∗
                                H_f + u] × F[j][r][u][v]
```

Vectorization and FMA instructions. We vectorize the basic im2win-based convolution algorithm using the Single Instruction Multiple Data (SIMD) instruction set and the FMA instructions. Each FMA instruction operates on N_{vec} scalar output elements simultaneously. In addition, there are a total of N_{reg} logical registers that can be addressed, so the number of elements that the registers can retain is $N_{reg}N_{vec}$. We used AVX2 instruction set in our implementation. This instruction set has 16 logical registers, each of which can store 256-bit of data. The most expensive computation in our algorithm is the AXPY operation (Line 9 in Algorithm 2), thus we implement it using the FMA instructions.

Loop reordering, hoist and loop unrolling. Algorithm 2 has seven nested for loops. Each for loop is independent of the others, hence we can arbitrarily change the order of the loops without affecting the computational results. Because the access costs of different dimensions of a tensor are different, to take advantage of the spatial locality of loading, we should prioritize the computation of dimensions that are less expensive to access and that exhibit consecutive memory access. Hence we reorder the loop structure of Algorithm 2 to the loop structure of Algorithm 3.

In Algorithm 2, $\hat{\mathcal{I}}$ is the most expensive tensor to access, so we should minimize the number of access to it. We observe that $\hat{\mathcal{I}}$ is independent of the C_o dimension of $\mathcal{O}$ and $\mathcal{F}$ (Line 9 in Algorithm 2), therefore we hoist the elements of $\hat{\mathcal{I}}$ to keep them in registers until they are no longer used (Line 10 - Line 14 in Algorithm 3). After loading the elements of $\mathcal{F}$ in the innermost loop (Line 19 in Algorithm 3), the FMA operation performs AXPY operation (Line 20 in Algorithm 3).

After determining the overall loop order of the algorithm, we consider the unrolling of individual loops. Loop unrolling reduces the number of cache/memory read, and also reduces the number of mis-branch predictions generated by the processor in each iteration. We observe that the innermost loop in Algorithm 2 has loop independence, hence we unroll (also

Algorithm 3: High Performance im2win-based Convolution Algorithm

Input: Input tensor $\mathcal{I}$, Filter tensor $\mathcal{F}$, Stride s
Output: Output tensor $\mathcal{O}$

1 $\hat{\mathcal{I}}[N_i][C_i][H_o][W_i \times H_f]$ = **Function** IM2WIN($\mathcal{I}, \mathcal{F}, s$)
2 **for** $jj = 0$ **to** $C_o/C_{o,b} - 1$ **in parallel do**
3 **for** $i = 0$ **to** $N_o - 1$ **do**
4 **for** $m = 0$ **to** $H_o - 1$ **do**
5 **for** $r = 0$ **to** $C_f - 1$ **do**
6 **for** $nn = 0$ **to** $W_o/W_{o,b} - 1$ **do**
7 **for** $vv = 0$ **to** $W_f/W_{f,b} - 1$ **do**
8 DOT_PRODUCT(jj, i, r, m, nn, vv, s)
9 **Function** DOT_PRODUCT(jj, i, r, m, nn, vv, s):
10 **for** $n' = 0$ **to** $W_{o,b} - 1$ **do**
11 **for** $v' = 0$ **to** $W_{f,b} - 1$ **do**
12 **for** $u = 0$ **to** $H_f/N_{vec} - 1$ **do**
13 $iu = (nn * W_{o,b} + n') * s * W_f + (vv * W_{f,b} + v') * H_f + u$
14 SIMD_LOAD($\hat{\mathcal{I}}[i][r][m][iu : iu+N_{vec}]$)
15 **for** $j' = 0$ **to** $C_{o,b} - 1$ **do**
16 SIMD_LOAD($\mathcal{O}[i][jj * C_{o,b} + j'][m][nn]$)
17 **for** $v' = 0$ **to** $W_{f,b} - 1$ **do**
18 **for** $u = 0$ **to** $H_f/N_{vec} - 1$ **do**
19 SIMD_LOAD($\mathcal{F}[jj * C_{o,b} + j'][r][vv * W_{f,b} + v'][u : u + N_{vec}]$)
20 FMA($\hat{\mathcal{I}}, \mathcal{F}, \mathcal{O}[i][jj * C_{o,b} + j'][m][nn]$)
21 SIMD_STORE($\mathcal{O}[i][jj * C_{o,b} + j'][m][nn]$)

called as flatten) [16] the innermost loop (Line 19 - Line 20 in Algorithm 3) in our implementation.

Register and cache blocking. The minimum number of output elements θ required to maintain the peak performance of the SIMD architecture system is limited by the maximum number of elements that can be kept in all vector registers [17], i.e., $\theta \leq N_{reg}N_{vec}$. To achieve maximum performance of our algorithm, we want to keep as many elements of tensors as possible in registers and minimize the number of SIMD load. Therefore, we adopt register blocking [18] to load elements of several dot product windows into registers (dynamically calculated based on W_f, H_f and N_{vec}) at Line 14 and Line 19 respectively in Algorithm 3. We apply register blocking to W_o and W_f dimensions at Line 10 - Line 11 in Algorithm 3 (omitted due to space limit), followed by a SIMD_LOAD in the filter height dimension H_f at Line 12 in Algorithm 3.

At the cache level, we can partition the input data to fit the cache size to minimize cache misses. Since our operation process is to continuously iterate a new filter window in the C_o dimension of output and perform dot product with $\hat{\mathcal{I}}$, thus improving the data reusability of the elements in $\hat{\mathcal{I}}$. We choose to divide the iterative output channel dimension C_o into smaller partitions to fit the memory to the next level of the hierarchy (Line 15 in Algorithm 3).

Parallelization strategy. We observe in our algorithm that all the output elements are independent of each other and are embarrassingly parallelizable [19]. Since the output is a four-dimensional tensor $\mathcal{O}(N_i \times C_o \times H_o \times W_o)$, this means that we can pick any one of four dimensions for parallelization. Our im2win-based convolution implementation extracts parallelism in the output channel C_o dimension. Each thread is assigned a block of output elements of size $H_o \times W_o \times C_o/t$, where t is the number of threads used.

IV. Experimental Results

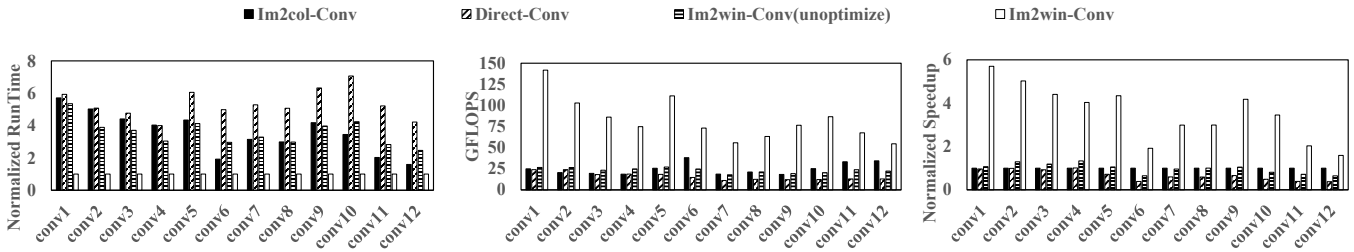
In this section, we compare the performance results and memory usages of our im2win-based convolution algorithms with Pytorch's im2col-based algorithm and a direct convolution implementation.

A. Experimental Setup

Platform. We run the experiments on two Intel Xeon Silver 4214 processors at 2.20GHz (each has 12 cores). Each CPU has 32KB L1 cache, 1MB L2 cache, and 16.5MB L3 cache.
Software. We implement our im2win-based convolution algorithm in C++ and parallelize it using OpenMP 4.5. We use the tensor data structure of PyTorch 1.10.0a0 [20], and the element in each tensor is a single-precision number. We compare our convolution algorithms with PyTorch's im2col-based convolution and direct convolution in terms of performance and memory usage. The im2col-based convolution algorithm in PyTorch is parallelized by OpenMP. The direct convolution code implemented in this work [7] is not open-sourced and uses a special data layout, so we implemented a naive direct convolution in C++ using nested loops and parallelized with OpenMP. We compile all the algorithms with GCC 7.5 and "-Ofast -march=native" compilation options.
Benchmarks. We want to cover the majority of the convolutional layers used in commonly used DNNs. However, it is impossible if we benchmark with one neural network model with just one filter size for all convolutional layers, for example, all filter tensors in VGG-16 [21] are 3×3, and ResNet-50 [22] contains only three different filters in sizes. Hence for our experimental evaluation, we select an state-of-the-art DNN benchmark [13], which includes twelve unique convolution layers, Conv1-Conv12 (shown in Table I).

Table I: Twelve convolution layers of the DNN benchmarks.

NAME	INPUT $C_i \times H_i \times W_i$	FILTER, STRIDE $C_o \times H_f \times W_f, s_h(s_w)$	OUTPUT $C_o \times H_o \times W_o$
Conv1	$3 \times 227 \times 227$	$96 \times 11 \times 11, 4$	$96 \times 55 \times 55$
Conv2	$3 \times 231 \times 231$	$96 \times 11 \times 11, 4$	$96 \times 56 \times 56$
Conv3	$3 \times 227 \times 227$	$64 \times 7 \times 7, 2$	$64 \times 111 \times 111$
Conv4	$64 \times 224 \times 224$	$64 \times 7 \times 7, 2$	$64 \times 109 \times 109$
Conv5	$96 \times 24 \times 24$	$256 \times 5 \times 5, 1$	$256 \times 20 \times 20$
Conv6	$256 \times 12 \times 12$	$512 \times 3 \times 3, 1$	$512 \times 10 \times 10$
Conv7	$3 \times 224 \times 224$	$64 \times 3 \times 3, 1$	$64 \times 222 \times 222$
Conv8	$64 \times 112 \times 112$	$128 \times 3 \times 3, 1$	$128 \times 110 \times 110$
Conv9	$64 \times 56 \times 56$	$64 \times 3 \times 3, 1$	$64 \times 54 \times 54$
Conv10	$128 \times 28 \times 28$	$128 \times 3 \times 3, 1$	$128 \times 26 \times 26$
Conv11	$256 \times 14 \times 14$	$256 \times 3 \times 3, 1$	$256 \times 12 \times 12$
Conv12	$512 \times 7 \times 7$	$512 \times 3 \times 3, 1$	$512 \times 5 \times 5$



(b) Normalized Runtime (c) GFLOPS (d) Normalized Speedup

Fig. 4: Performance comparison of our convolution algorithms with PyTorch's im2col-based convolution and direct convolution. The runtime results are normalized to our convolution, and the performance results in GFLOPS are normalized to im2col-based convolution. Among twelve convolution layers, our convolution has the fastest runtime, while PyTorch's runtime is average to 3.6× slower and the direct convolution's runtime is average to 5.3× slower than us. Our convolution has up to about 3.6× GFLOPS than that of Pytorch and 5.3× that of direct convolution, respectively.

B. Performance

In the following experiments, we use the wall-clock time in the standard C++ library to measure the runtime of different algorithms. We run each algorithm five times and record the best runtime among five runs. The batch size of each convolution layer input data is 128.

Figure 4(b) shows the normalized runtime of the different convolution algorithms. The runtime results are normalized to our optimized im2win convolution algorithm. Overall, our algorithm has the shortest runtime among all twelve different convolutional layers. Pytorch is up to 5.7× slower and the direct convolution is up to 7× slower than our algorithm, respectively. Shown in Figure 4(c), We measure the floating-point computational performance of different convolutional algorithms, and normalize the results to our algorithm as Figure 4(d). Our convolution has at most 6× GFLOPS than PyTorch, and at least 2× GFLOPS than PyTorch. Our convolution has a maximum of about 7× GFLOPS and a minimum of about 4× GFLOPS than direct convolution.

C. Memory Usage

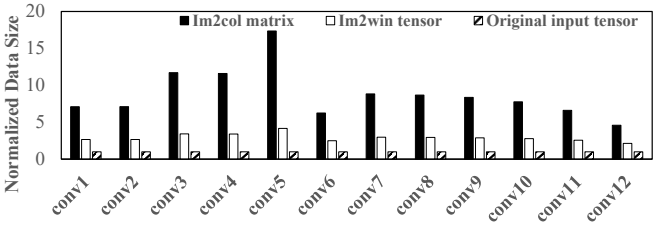


Fig. 5: Normalized data sizes of the im2col matrix, the im2win tensor and the original input tensor of twelve convolution layers. We normalize all the data sizes to the original input tensor.

We show the data sizes of the im2col matrix, the im2win tensor and the original input tensor of twelve convolution layers as Figure 5 and the peak memory usage results of all convolution algorithms as Figure 6. On average, the im2col matrices are 2.93× larger than our im2win tensors for all twelve different convolutional layers. The im2col-based convolution uses up to 2.75× more memory than our im2win-based convolution. Our im2win-based convolution reduces memory overhead by average to 41.6% compared to the im2col-based convolution algorithm. The results show that our convolution algorithm can more effectively reduce the memory footprint compared to the im2col-based convolution algorithm.

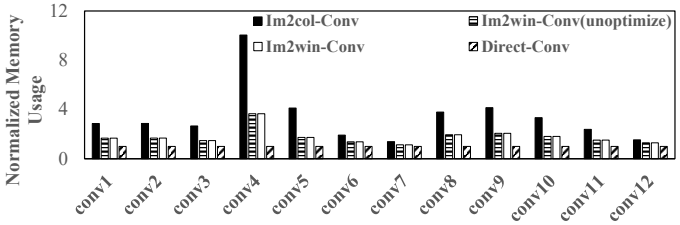


Fig. 6: Normalized peak memory usage of all convolution algorithms on twelve convolution layers. Since direct convolution has no additional memory overhead of the data transformation, we normalize all the memory usage results to direct convolution.

V. Conclusion

We proposed a memory-efficient im2win algorithm. Different from the classic im2col algorithm, im2win refactorizes a row of square or rectangle dot product windows of the input image and flattens unique elements within these windows into a row in the output tensor, which enables consecutive memory access and data reuse, and thus greatly reduces the memory overhead. In addition, we proposed a high-performance im2win-based convolution algorithm, which implements a nested loop structure similar to direct convolution to improve data reusability and parallel performance. We tested on a benchmark containing most of the convolution layers in a neural network model. Experimental results show that our algorithm reduces memory overhead by average to 41.6% compared to PyTorch's im2col-based convolution algorithm, and achieves speedups in performance of average to 3.6× and 5.3× compared to PyTorch's im2col-based convolution algorithm and direct convolution algorithm, respectively.

Acknowledgment

This work was supported by National Natural Science Foundation of China (Grant No. 62162045), Key Re-

search and Development Program of Jiangxi (Program No. 20192BBE50073), and Technology Innovation Guidance Program Project of Jiangxi Province (Special Project of Technology Cooperation) (Grant No. 20212BDH81003).

REFERENCES

[1] E. J. Crowley, G. Gray, and A. J. Storkey, "Moonshine: Distilling with cheap convolutions," *Advances in Neural Information Processing Systems*, vol. 31, 2018.

[2] N. Ma, X. Zhang, H.-T. Zheng, and J. Sun, "Shufflenet v2: Practical guidelines for efficient cnn architecture design," in *Proceedings of the European conference on computer vision (ECCV)*, 2018, pp. 116–131.

[3] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, 2017.

[4] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, 2019.

[5] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "TensorFlow: A system for Large-scale machine learning," in *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, 2016, pp. 265–283.

[6] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. S. Duff, "A set of level 3 basic linear algebra subprograms," *ACM Transactions on Mathematical Software (TOMS)*, vol. 16, no. 1, pp. 1–17, 1990.

[7] J. Zhang, F. Franchetti, and T. M. Low, "High performance zero-memory overhead direct convolutions," in *International Conference on Machine Learning*. PMLR, 2018, pp. 5776–5785.

[8] K. Chellapilla, S. Puri, and P. Simard, "High performance convolutional neural networks for document processing," in *Tenth International Workshop on Frontiers in Handwriting Recognition*. Suvisoft, 2006.

[9] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *Proceedings of the 22nd ACM international conference on Multimedia*, 2014, pp. 675–678.

[10] J. A. Gunnels, G. M. Henry, and R. A. Geijn, "A family of high-performance matrix multiplication algorithms," in *International Conference on Computational Science*. Springer, 2001, pp. 51–60.

[11] A. Anderson, A. Vasudevan, C. Keane, and D. Gregg, "Low-memory gemm-based convolution algorithms for deep neural networks," *CoRR*, vol. abs/1709.03395, 2017.

[12] A. V. Trusov, E. E. Limonova, D. P. Nikolaev, and V. V. Arlazarov, "p-im2col: Simple yet efficient convolution algorithm with flexibly controlled memory overhead," *IEEE Access*, vol. 9, pp. 168 162–168 184, 2021.

[13] M. Cho and D. Brand, "Mec: memory-efficient convolution for deep neural network," in *International Conference on Machine Learning*. PMLR, 2017, pp. 815–824.

[14] A. Heinecke, G. Henry, M. Hutchinson, and H. Pabst, "Libxsmm: accelerating small matrix multiplications by runtime code generation," in *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2016, pp. 981–991.

[15] E. Georganas, S. Avancha, K. Banerjee, D. Kalamkar, G. Henry, H. Pabst, and A. Heinecke, "Anatomy of high-performance deep learning convolutions on simd architectures," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2018, pp. 830–841.

[16] J. J. Dongarra and A. Hinds, "Unrolling loops in fortran," *Software: Practice and Experience*, vol. 9, no. 3, pp. 219–226, 1979.

[17] T. M. Low, F. D. Igual, T. M. Smith, and E. S. Quintana-Orti, "Analytical modeling is enough for high-performance blis," *ACM Transactions on Mathematical Software (TOMS)*, vol. 43, no. 2, pp. 1–18, 2016.

[18] M. Wolfe, "Iteration space tiling for memory hierarchies," in *Proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing*. USA: Society for Industrial and Applied Mathematics, 1989, p. 357–361.

[19] M. Herlihy, N. Shavit, V. Luchangco, and M. Spear, *The art of multiprocessor programming*. Newnes, 2020.

[20] "PyTorch," 2022. [Online]. Available: https://github.com/pytorch/pytorch

[21] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *CoRR*, vol. abs/1409.1556, 2015.

[22] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.