Im2win: An Efficient Convolution Paradigm on GPU

Shuai Lu[1], Jun Chu[1], Luanzheng Guo[2], and Xu T. Liu[3][0000−0003−3980−9803]

[1] Nanchang Hangkong University, Nanchang, Jiangxi Province, China
2016085400101@stu.nchu.edu.cn, chuj@nchu.edu.cn
[2] University of California Merced, California, USA
lguo4@ucmerced.edu
[3] University of Washington, Seattle, Washington, USA
x0@uw.edu

Abstract. Convolution is the most time-consuming operation in deep neural network operations, so its performance is critical to the overall performance of the neural network. The commonly used methods for convolution on GPU include the general matrix multiplication (GEMM)-based convolution and the direct convolution. GEMM-based convolution relies on the im2col algorithm, which results in a large memory footprint and reduced performance. Direct convolution does not have the large memory footprint problem, but the performance is not on par with GEMM-based approach because of the discontinuous memory access. This paper proposes a window-order-based convolution paradigm on GPU, called `im2win`, which not only reduces memory footprint but also offers continuous memory accesses, resulting in improved performance. Furthermore, we apply a range of optimization techniques on the convolution CUDA kernel, including shared memory, tiling, micro-kernel, double buffer, and prefetching. We compare our implementation with the direct convolution, and PyTorch's GEMM-based convolution with cuBLAS and six cuDNN-based convolution implementations, with twelve state-of-the-art DNN benchmarks. The experimental results show that our implementation 1) uses less memory footprint by 23.1% and achieves 3.5× TFLOPS compared with cuBLAS, 2) uses less memory footprint by 32.8% and achieves up to 1.8× TFLOPS compared with the best performant convolutions in cuDNN, and 3) achieves up to 155× TFLOPS compared with the direct convolution. We further perform an ablation study on the applied optimization techniques and find that the micro-kernel has the greatest positive impact on performance.

Keywords: Convolution · CUDA · im2win · im2col · parallel computing · CNN

1 Introduction

Convolutional neural network (CNN) is an important network model widely used in computer vision, image processing, and scientific computing. CNN consists of an input layer, an output layer, and convolutional layers between them [7]. Convolutional operations can take 50% - 90% of the total inference operations of the neural network model [15]. Also, convolution operations often account for over 90% of the total execution time of many neural networks [20]. Therefore, it is critical to reduce the cost of convolutional operations to improve the overall performance of neural networks.

Graphics processing unit (GPU) has been used to accelerate tensor convolution operations. Popular deep learning frameworks, such as PyTorch [17] and TensorFlow [3], use GPU to accelerate convolution operations with cuBLAS [1] and cuDNN [5], both developed by NVIDIA. cuBLAS is a GPU-accelerated library for the basic linear algebra subroutines. cuDNN is a set of primitives for forward and backward convolution, pooling, normalization, and activation layers used by neural networks.

There are mainly two types of convolution methods on GPU in terms of data transformation: the im2col data transformation-based and no data transformation at all. The im2col-based convolution transforms the input tensor and the filter tensor into two matrices, known as *the im2col algorithm*, followed by the general matrix-matrix multiplication (GEMM) with cuBLAS or cuDNN, and finally transforms the resultant matrix back to the output tensor [4]. The problem with the im2col-based convolution is that 1) the im2col operation generates a high memory footprint and bandwidth overhead, which is exaggerated on GPU where the memory/cache capacity is highly limited; 2) its performance is significantly affected by the performance of the GEMM operation in cuBLAS, which takes the input im2col matrix and the filter im2col matrix as inputs while the two matrices are significantly different in size, leading to bad performance [22,9].

A typical direct convolution has no data transformation, and is implemented as seven nested for loops over the original input and filter tensors, with the scalar a multiplied by x plus y (AXPY) computed in the innermost loop [22]. Compared to the im2col-based convolution, the direct convolution has no additional memory overhead. However, its AXPY operations suffer from discontinuous memory access, because of visiting distinct dimensions of the input tensor across the nested for loops. This results in low data reuse and low cache hit rate. This problem is seriously magnified on GPU.

To solve similar problems on CPU, we previously proposed a novel convolution algorithm, called `im2win` [14] (image to window), which rearranges the input tensor in the access order of the dot product windows. In this paper, we evolve the `im2win` algorithm and develop a memory-efficient and high-performance `im2win`-based convolution paradigm on GPU. The `im2win` convolution paradigm first transforms the input tensor into an `im2win` tensor using the `im2win` data transformation (see Section 3.1). Next, the convolution is implemented as a three-level nested loop structure akin to an implicit GEMM convolution, and the indices of input tensor, filter tensor and output tensor can be mapped tothe three levels of for loops when performing an AXPY operation. Our `im2win` data transformation algorithm can significantly reduce memory consumption compared to the im2col data transformation. We implement the `im2win`-based convolution paradigm in CUDA and propose a range of optimization techniques, including tiling, micro-kernel, double buffer, and prefetching.

We compare our implementation with various convolution methods, including the direct convolution, PyTorch's GEMM-based convolution using cuBLAS, and six different cuDNN-based convolution implementations, using twelve different state-of-the-art deep neural network benchmarks. The results of our experiments indicate that our implementation outperforms the others in different aspects. Specifically, it uses less memory by 23.1% compared to cuBLAS and by 32.8% compared to the best-performing convolution implementations in cuDNN, while on average achieving 3.5× and 1.1× TFLOPS, respectively. Additionally, our implementation achieves up to 155× TFLOPS

compared with the direct convolution. We also conduct an ablation study to understand which optimization technique has the greatest positive impact on performance, and find that the micro-kernel has the most significant effect. We make our code publicly available at https://github.com/seth-lu/Im2win under *cuda* branch.

To summarize, this paper makes the following **contributions**:

1) We propose an innovative convolution paradigm on GPU, called `im2win`-based convolution (Section 3.2), along with a set of optimizations that are specifically designed to improve its memory efficiency and performance (Section 3.3). Our proposed convolution paradigm is shown to be both high-performance and memory-efficient, offering a promising alternative to existing convolution methods on GPU.

2) We implement our `im2win`-based convolution in CUDA and compare it with the direct convolution, existing convolution algorithms in cuBLAS and cuDNN. We conduct an experimental evaluation using twelve DNN benchmarks of various dimensions that provides a comprehensive result of our proposed method (Section 4.2).

3) We conduct an ablation study on the optimization techniques applied to the proposed im2win-based convolution paradigm, which reveals that the micro-kernel optimization technique has the most significant impact on performance (see Section 4.4).

The rest of paper is organized as follow. Section 2 defines the notations used in this paper, reviews existing convolution techniques and related works. Section 3 presents our convolution paradigm on GPU along with a set of optimizations that are specifically designed to improve its memory efficiency and performance. We conduct an experimental evaluation, an ablation study and present the performance and memory usage of different convolution algorithms in Section 4. Finally, we conclude our work in Section 5.

2 Preliminaries and Related Work

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

2.1 Notations

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:

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

subject to

$$\begin{aligned} &i = 0, 1, .., N_i - 1, j = 0, 1, .., C_o - 1, m = 0, 1, .., H_o - 1, \\ &n = 0, 1, .., W_o - 1, u = 0, 1, .., H_f - 1, v = 0, 1, .., W_f - 1, \\ &r = 0, 1, .., C_i - 1. \end{aligned}$$

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.

2.2 The direct convolution

The direct convolution is one of the most naive implementations of convolutions. 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 $\mathcal{I}$. Each element of $\mathcal{O}$ is computed with an AXPY operation in the innermost loop. These nested loops can be parallelized well on GPU. However, the larger $\mathcal{O}$ is, the less data can fit in the cache. In this case, the direct convolution accesses directly through the global memory. The data access is discontinuous and the latency is high, resulting in poor performance [22]. It has been shown that the performance of the direct convolution can be greatly improved by redesigning specific data layouts and data flows on the GPU [19].

2.3 The GEMM-based convolution

The GEMM-based convolution proposed by Chellapilla et al. [4] is the most commonly used convolution algorithm, and is widely used in existing deep learning frameworks [17,3]. Due to its fundamental and general nature, it is often used as a benchmark for performance comparison. The GEMM-based convolution unrolls the convolution operation into a GEMM 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 on the right in Figure 1, 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 transformed from R': $R(j, m, n) = R'(mW_o + n, j)$.

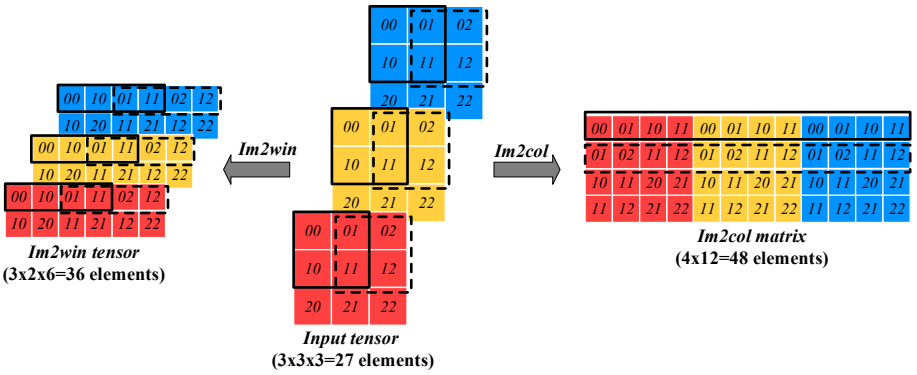

Fig. 1: The im2col and im2win data transformation examples with $C_i = H_i = W_i = 3$, $H_f = W_f = 2$, $s_h = s_w = 1$. The solid and dashed boxes indicate the different dot product windows of the input tensor. We can see that the im2win tensor has less elements than the im2col matrix.

Dongarra et al. has demonstrated that the GEMM-based convolution benefits from the efficient implementation on GPU and the nature of GPU architectures [8]. Due to the highly optimized cuBLAS library, GEMM-based convolution has reliable performance and supports various input tensor sizes. However, this approach requires a large memory to store the im2col matrix transformed from the input tensor and the filter tensor. Because it has to store duplicated elements due to overlap of the filter positions in the convolution, the im2col matrix is much larger than the original tensor. What's worse, the im2col matrix is much larger than the filter matrix, this results the GEMM operation in significantly lower performance than the best achievable performance [22,9]. MEC proposes a compact lowering trick on the im2col matrix and splits a single GEMM into multiple small GEMMs to reduce the memory footprint [6]. The small GEMM operations are performed in parallel to complete the convolution. We intend to compare our convolution with MEC on GPU, unfortunately, MEC is not open-sourced.

2.4 The convolution algorithms implemented in cuDNN

cuDNN is a GPU-accelerated deep learning library from NVIDIA, which implements six convolution algorithms including the *direct convolution*, the *GEMM-based convolution*, two *implicit GEMM-based convolutions*, the *Fast Fourier Transform (FFT) convolution*, and the *Winograd convolution*. The implicit GEMM-based convolution is a variant of the direct convolution, which operates natively on the input tensors, converting the computation into a GEMM on the fly. During the computation, the im2col matrices are implicitly formed. There is another variant that precomputes offsets used in the implicit GEMM. The FFT convolution uses the fast Fourier transform to achieve convolution. It can achieve fast convolution with fewer operations the direct convolution, however, it requires more memory and is more difficult to implement as it works with complex numbers instead of real numbers. The Winograd convolution is based on the Winograd's minimal filtering algorithm [13], which is computationally efficient for some small convolution kernels.

cuDNN supports autotunning, which automatically selects an algorithm on a per-layer basis based on the layer dimensions. But even so, cuDNN still has some shortcomings. The cuDNN call parameter API is pre-defined, so it does not have the flexibility to build some special convolutions. cuDNN often resorts to a slower algorithm that fits the workspace size constraint. To alleviate this behavior of cuDNN, u-cuDNN divides layers' mini-batch computation into multiple micro-batches transparently by decreasing the workspace size requirements [16]. We refer the readers the performance evaluation of cuDNN convolution algorithms in [12].

3 The im2win-based convolution paradigm on GPU

To reduce the huge memory usage of the im2col-based convolution and avoid nonconsecutive memory access of the direct convolution, we use the `im2win` data transformation and propose a high-performance and memory efficient `im2win`-based convolution paradigm on GPU. Furthermore, we propose several optimization techniques for our `im2win`-based convolution.

3.1 Motivations

Now we present the `im2win` data transformation algorithm and the implicit GEMM-based convolution algorithm as the motivations of our `im2win`-based convolution.

The `im2win` data transformation algorithm. As shown on the left in Figure 1, our image to window algorithm (called `im2win`) rearranges the input tensor $\mathcal{I}$ in the access order of the dot product windows. It dramatically reduces memory overhead with more compact data arrangement compared with the im2col data transformation algorithm. For the dot product windows in the same row, each dot product operation reuses the elements of the previous loaded window except for the first one. Our im2win algorithm supports great data reusability, temporal and spatial data locality.

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 1). 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:

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

subject to

$$\begin{aligned} m &= 0, 1, .., H_o - 1, n = 0, 1, .., W_o - 1, u = 0, 1, .., H_f - 1, \\ v &= 0, 1, .., W_f - 1, i = 0, 1, .., N_i - 1, r = 0, 1, .., C_i - 1, \\ k &= 0, 1, .., W_i - 1. \end{aligned}$$

Recall in Figure 1 $s = 1$, the im2col matrix has 48 elements, while in Figure 1, the im2win tensor has 36 elements. The im2win tensor has 1/3 less elements than the im2col matrix in addition to provide better data locality and data reusability.

The implicit GEMM-based convolution algorithm. In addition to the GEMM-based convolution algorithm with explicit im2col data transformation, there is also an implicit GEMM-based convolution algorithm, shown in Algorithm 1. Instead of an explicit data transformation process, a three-level nested for loop structure is used in the algorithm to calculate the indices of $\mathcal{I}$ (Line 4 and Line 8 - Line 13), $\mathcal{F}$ (Line 2 and Line 8 - Line 11) and $\mathcal{O}$ (Line 2 - Line 6). In the innermost loop, the AXPY operation is performed to result in $\mathcal{O}$ (Line 14). Implicit GEMM-based convolution does not have the memory consumption of data transformation. The name of implicit GEMM-based convolution algorithm can be confusing. With no explicit input and filter matrices, it is impossible to call cuBLAS GEMM API. In addition, the indices to perform an AXPY must be computed on the fly. This algorithm is commonly viewed as a variant of the direct convolution. Since Algorithm 1 has the same three-level nested for loop structure as GEMM operation, the optimization techniques that are proposed for GEMM can also be applied to implicit GEMM-based convolution algorithm, such as shared memory, tiling, micro-kernel, vectorized load/store and prefetching.

Algorithm 1: Implicit GEMM-based Convolution Algorithm

Input: Input $\mathcal{I}$, Filter $\mathcal{F}$, Stride s
Output: Output O
Dimensions: $\mathbf{M} = C_o, \mathbf{N} = N_o \times H_o \times W_o, \mathbf{K} = C_f \times H_f \times W_f$

1 **for** $m = 0$ **to** $M - 1$ **do**
2 $o_c = f_n = m$
3 **for** $n = 0$ **to** $N - 1$ **do**
4 $o_n = i_n = n/(H_o \times W_o)$
5 $o_h = (n\%(H_o \times W_o))/W_o$
6 $o_w = (n\%(H_o \times W_o))\%W_o$
7 **for** $k = 0$ **to** $K - 1$ **do**
8 $f_c = i_c = k/(H_f \times W_f)$
9 $k_{res} = k\%(H_f \times W_f)$
10 $f_h = k_{res}/W_f$
11 $f_w = k_{res}\%W_f$
12 $i_h = o_h \times s + f_h$
13 $i_w = o_w \times s + f_w$
14 $O(o_n, o_c, o_h, o_w) += \mathcal{I}(i_n, i_c, i_h, i_w) \times \mathcal{F}(f_n, f_c, f_h, f_w)$

Algorithm 2: Basic Im2win-based Convolution On GPU

Input: Input $\mathcal{I}$, Filter $\mathcal{F}$, Stride s
Output: Output O
Im2winTensor: $\hat{\mathcal{I}}$ = **Function** IM2WIN($\mathcal{I}, \mathcal{F}, s$)
Dimensions : $\mathbf{M} = C_o, \mathbf{N} = N_o \times H_o \times W_o, \mathbf{K} = C_f \times H_f \times W_f$
# of blocks : $M/32 \times N/32$
# of threads per block: 32×32

1 $m = bx \times 32 + tx, n = by \times 32 + ty$
2 $o_c = m, o_n = i_n = n/(H_o \times W_o)$
3 $o_h = (n\%(H_o \times W_o))/W_o$
4 $o_w = (n\%(H_o \times W_o))\%W_o$
5 **for** $k = 0$ **to** $K - 1$ **do**
6 $f_c = i_c = k/(H_f \times W_f)$
7 $k_{res} = k\%(H_f \times W_f)$
8 $f_h = k_{res}/W_f, f_w = k_{res}\%W_f$
9 $i_h = o_h \times s + f_h, i_w = o_w \times s + f_w$
10 $O(o_n, o_c, o_h, o_w) += \hat{\mathcal{I}}(i_n, i_c, i_h, i_w) \times \mathcal{F}(f_n, f_c, f_h, f_w)$

3.2 The `im2win`-based convolution on GPU

We propose a basic `im2win`-based convolution on GPU shown in Algorithm 2 implemented in CUDA. The input tensor $\mathcal{I}$ is initially transformed into the `im2win` tensor $\hat{\mathcal{I}}$ based on Equation (2). Next, the convolution is implemented as a three-level nested for loop structure same as the implicit GEMM-based convolution. In Algorithm 2, dimension M and dimension N are mapped to grid and block respectively, where each block includes 32x32 threads, i.e., grid = (M/32, N/32), block = (32, 32). The bx and by denote block indices in the x and y dimensions respectively, and tx and ty denote thread indices in the x and y dimensions respectively (Line 1). Within the kernel of each block, the three levels of for loops are $\mathbf{M} = C_o, \mathbf{N} = N_o \times H_o \times W_o$, and $\mathbf{K} = C_f \times H_f \times W_f$. The indices of $\hat{\mathcal{I}}$ (Line 2 , Line 6 -Line 9), $\mathcal{F}$ (Line 6 -Line 8) and O (Line 2 -Line 4) tensor are computed on the fly within the kernel function. The innermost for loop performs an AXPY operation.

In the kernel function, we first compute indices m and n from dimension M and dimension N respectively from the global indices of the thread tx and ty (Line 1 in Algorithm 2). Next, the indices of the four dimensions of the output tensor required for the AXPY operation are calculated by performing division and remainder operations on m and n (Line 2 - Line 4). Finally, we compute the remaining indices of $\hat{\mathcal{I}}$ and $\mathcal{F}$ in a for loop in dimension K, and perform AXPY operations after obtaining all the indices of O, $\hat{\mathcal{I}}$ and $\mathcal{F}$ (Line 5 - Line 10).

The most expensive computation in Algorithm 2 is the AXPY operation at Line 10, which requires three read operations and one write operation. On GPU, frequent read

Algorithm 3: High Performance Im2win Convolution Algorithm On GPU

Input: Input tensor $\mathcal{I}$, Filter tensor $\mathcal{F}$, Stride s
Output: Output tensor $\mathcal{O}$
Im2winTensor: $\hat{\mathcal{I}}$ = **Function** IM2WIN($\mathcal{I}, \mathcal{F}, s$)
Dimensions : $\mathbf{M} = C_o, \mathbf{N} = N_o \times H_o \times W_o, \mathbf{K} = C_f \times H_f \times W_f$
# of blocks : $M/M_B \times N/N_B$
# of threads per block: $M_B/M_T \times N_B/N_T$
1 Registers: $R_{\hat{\mathcal{I}}}[2][N_T], R_{\mathcal{F}}[2][M_T], R_O[M_T \times M_T]$ *//double buffer*
2 Shared memories: $S_{\hat{\mathcal{I}}}[2][K_B \times N_B], S_{\mathcal{F}}[2][M_B \times K_B]$ *//double buffer*
3 $S_{\hat{\mathcal{I}}}[0][k_B \times n_B] \underleftarrow{\text{load}}\ k_B \times n_B$ of $\hat{\mathcal{I}}(0, by)$
4 $S_{\mathcal{F}}[0][m_B \times k_B] \underleftarrow{\text{load}}\ m_B \times k_B$ of $\mathcal{F}(bx, 0)$
5 __syncthreads()
6 $R_{\hat{\mathcal{I}}}[0][n_T] \underleftarrow{\text{vec\_load}}\ n_T$ of $S_{\hat{\mathcal{I}}}[0][0 \times n_B]$
7 $R_{\mathcal{F}}[0][m_T] \underleftarrow{\text{vec\_load}}\ m_T$ of $S_{\mathcal{F}}[0][m_B \times 0]$
8 **for** $kk = 0$ **to** $C_f \times H_f \times W_f/K_{f,b} - 1$ **do**
9 **for** $k' = 1$ **to** $K_{f,b} - 1$ **do**
10 $R_{\hat{\mathcal{I}}}[load][n_T] \underleftarrow{\text{vec\_load}}\ n_T$ of $S_{\hat{\mathcal{I}}}[store][k' \times n_B]$ *//prefetching*
11 $R_{\mathcal{F}}[load][m_T] \underleftarrow{\text{vec\_load}}\ m_T$ of $S_{\mathcal{F}}[store][m_B \times k']$ *//prefetching*
12 $R_O[m_T \times n_T]$ += $R_{\mathcal{F}}[store][m_T] \times R_{\hat{\mathcal{I}}}[store][n_T]$ *//micro-kernel*
13 **if** $kk \neq C_f \times H_f \times W_f/K_{f,b} - 1$ **then**
14 $S_{\hat{\mathcal{I}}}[load][k_B \times n_B] \underleftarrow{\text{load}}\ k_B \times n_B$ of $\hat{\mathcal{I}}(kk + 1, by)$ *//prefetching*
15 $S_{\mathcal{F}}[load][m_B \times k_B] \underleftarrow{\text{load}}\ m_B \times k_B$ of $\mathcal{F}(bx, kk + 1)$ *//prefetching*
16 __syncthreads()
17 $R_{\hat{\mathcal{I}}}[0][n_T] \underleftarrow{\text{vec\_load}}\ n_T$ of $S_{\hat{\mathcal{I}}}[store][0 \times n_B]$
18 $R_{\mathcal{F}}[0][m_T] \underleftarrow{\text{vec\_load}}\ m_T$ of $S_{\mathcal{F}}[store][m_B \times 0]$
19 $R_O[m_T \times n_T]$ += $R_{\hat{\mathcal{I}}}[1][n_T] \times R_{\hat{\mathcal{I}}}[1][n_T]$ *//micro-kernel*
20 $O(bx, by) \underleftarrow{\text{store}}\ R_O[m_T \times n_T]$

and write operations to the global memory have substantial latency. Therefore we need to cache as much data as possible used for AXPY operations into shared memory and registers per block, which have much lower latency. At Line 2 - Line 4 of the algorithm, we divide the index of outputs based on the global id of the thread so that each individual thread is responsible for a separate output. This data partition is obvious, but not computationally efficient. We can use the micro-kernel technique (elaborated shortly) to partition the $M_T \times N_T$ of O computation tasks for each individual thread, which will increase the data reusability. We propose in the next subsection a composition of optimizations making the best use of the `im2win`-based convolution on GPU.

3.3 Optimizations on GPU

Inspired by the optimization techniques used in GEMM on GPU, we apply the following optimizations to Algorithm 2, including tiling, shared memory, micro-kernel, vec-

torized load/store, double buffer, and prefetching. Those optimizations are especially important to maximize workload and data parallelism and reduce data access latency. We present our high-performance `im2win`-based convolution on GPU as Algorithm 3.

Tiling. Since Algorithm 2 has a similar implicit GEMM-based convolution implementation with three nested loops of M, N and K, the indices of the input tensor can be divided into small blocks called *tiles* [21]. We tile the sizes of $\hat{\mathcal{I}}$ and $\mathcal{F}$ into sizes of $M_B \times N_B \times K_B$ at the block level and $M_T \times N_T$ at the thread level in Algorithm 3. As the basic computational unit during computation, the main effect of tiling is to improve computational performance by reducing data accesses and improving data locality. For example, the size of the tile can be adapted to match the size of the shared memory or the registers, which has substantial lower latency, to improve the data reuse and to increase cache hit rate.

Shared memory and register. The memory on a GPU device consists of four levels of hierarchy: the global memory, the shared memory, the L1&L2 caches (not programmable in CUDA) and the registers. From the global memory to the shared memory, and to the registers, the access latency decreases and the size also decreases. After tiling the input tensor and the filter tensor, we allocate registers and shared memory blocks of size $M_B \times K_B$ and size $K_B \times N_B$ (Line 1 - Line 2), and we load $\hat{\mathcal{I}}$ and $\mathcal{F}$ located in global memory into the registers and shared memory (Line 3 - Line 7) in Algorithm 3. Because each dot product operation reuses the elements of the previous loaded dot product window from the same row in the `im2win` tensor. To take advantage of this, we load the data to the share memory of each block with as many dot product windows from the same row as possible, to achieve highest possible data reusability and data locality.

Micro-kernel. The micro-kernel technique can be used to increase the computational intensity. Without it, one AXPY operation in the innermost for loop of our `im2win`-based convolution computes one element of O. Micro-kernel are typically implemented as outer product multiplications of vectors. With each micro-kernel used in each thread in a block, each thread is now responsible for computing multiple elements of O. We tile the size of the micro-kernel at $M_T \times N_T$ divided at the thread level (Line 12 - Line 19 in Algorithm 3). The micro-kernel partitions the matrix multiplication among multiple threads, reducing the number of memory accesses and improving the parallelism and computational efficiency of the AXPY operations.

Vectorized load/store. The vectorized load/store are techniques to improve memory access efficiency by loading or storing multiple consecutive data elements from the shared memory into registers under single instruction (SIMD), thereby improving data IO efficiency and memory throughput. Data IO and memory throughput are often the performance bottlenecks when performing convolutional computation on the GPU. Our im2win tensor data structure is stored in a consecutive physical memory, with the dot product windows of the same row arranged continuously. Because each APXY operation loads consecutive dot product windows in the micro-kernel, loading $\hat{\mathcal{I}}$ and $\mathcal{F}$ from the shared memory of a block into the registers can be done using vectorized load (Line 6 - Line 7 in Algorithm 3).

Double buffer and prefetching. The double buffer optimization refers to the use of two buffers to store the input and filter tensors for pipelined concurrent computation. In Algorithm 3, we allocate two registers at Line 1 and two shared memories at Line 2.

Typically, one buffer is used for the ongoing computation and the other is for prefetching the new data used into registers (or shared memory) in the next computation. It hides the latency and overhead of loading data. When the computation is completed, the roles of the two buffers are swapped, i.e., the original buffer becomes the new load buffer and the original load buffer becomes the new computation buffer. The prefetching technique is performed on $\hat{\mathcal{I}}$ and $\mathcal{F}$ (Line 10 - Line 11 and Line 14 - Line 15 in Algorithm 3), followed by a *__syncthread()* that synchronizes the data among all the threads of a block performing prefetching. The prefetching technique allows certain amount of data (we prefetch 128 elements for the shared memory, and 8 elements for the register in our implementation) to be prefetched before the computation, thus reducing data waiting time and improving computational efficiency [11].

4 Experimental Results

In this section, we compare our `im2win` convolution algorithm with a naive direct convolution, PyTorch's im2col-based algorithm using cuBLAS and cuDNN's convolution implementations, present the performance results and memory usages of them, and perform an ablation study of our proposed optimization techniques.

4.1 Experimental Setup

Platform. We perform our experiments on a NVIDIA GeForce RTX 3090 GPU which has 24GB memory and is connected to an Intel Xeon Silver 4214 CPU server.
Software. The APIs of cuBLAS and cuDNN are pre-defined and are not available for the `im2win`-based convolution, so we implement our `im2win` convolution paradigm using CUDA 11.1. We use the tensor data structure of PyTorch 1.10.0 [2] with the single 32bit precision. We list the algorithms we compared, theirs notations, and their descriptions in Table 1.

Table 1: The convolution algorithms used in the experimental evaluations, the notations used in figures, and their implementation details.

Notation	Description
im2col+cuBLAS	the im2col-based convolution in PyTorch using cuBLAS 11.2
direct	a naive direct convolution implemented in CUDA 11.1
cuDNN	six convolutions in PyTorch using cuDNN 8.0.1
im2winGPU	our im2win-based convolution implemented in CUDA 11.1

Benchmarks. We aim to check how well our convolution paradigm performs on various convolutional layers in terms of dimensions. However, it is not persuasive if we only benchmark with one neural network model. For example, all the filters in VGG-16 [18] are 3×3, and ResNet-50 [10] contains only three different filters in sizes. Hence we select twelve state-of-the-art DNN benchmarks [6] in our evalution, including twelve unique convolution layers, Conv1-Conv12 (the parameters are shown in Table 2).

Table 2: Parameters of the twelve 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	$512 \times 7 \times 7$	$512 \times 3 \times 3, 1$	$512 \times 5 \times 5$

Table 3: The fastest algorithms selected by cuDNN automatically on twelve benchmarks.

cuDNN ALGORITHM Fastest chosen
IMPLICIT_GEMM
IMPLICIT_GEMM
IMPLICIT_GEMM
IMPLICIT_GEMM
WINOGRAD
IMPLICIT_GEMM
IMPLICIT_GEMM
FFT
WINOGRAD
WINOGRAD
WINOGRAD
IMPLICIT_GEMM

4.2 Performance

In the experiments, we use the wall-clock time in the standard C++ library to measure the runtime of different algorithms. We run each algorithm 100 times and record the best runtime among 100 runs. The batch size of each benchmark input data is 128.

Figure 2 shows the TFLOPS of different convolution algorithms of twelve different DNN benchmarks respectively on GPU. cuDNN has six convolution algorithms, with the fastest automatically chosen based on the input tensor dimensions. Table 3 shows the fastest algorithm automatically chosen by cuDNN at different benchmarks. Among the twelve benchmarks, our im2win-based convolution achieves about on average 3.5 × TFLOPS than that of im2col+cuBLAS convolution, and achieves 5× to 155× TFLOPS compared with the direct convolution Our im2win-based convolution has comparable performance with the cuDNN convolutions and achieves up to 1.8 × TFLOPS (the first benchmark) than that of the fastest algorithm chosen by cuDNN. Thanks to our customized optimizations tailored for our im2win-based convolution on GPU, we demonstrate better performance than the im2col-based convolution and the direct convolution of cuDNN, and show comparable performance with the implicit GEMM-based convolution, the FFT convolution, and the Winograd convolution in cuDNN.

4.3 Memory Usage

Figure 3 shows the memory usages of different convolution algorithms on twelve different DNN benchmarks respectively on GPU. Note that cuDNN auto-tunes itself to use the fastest algorithms among its six convolution algorithms based on the input tensor dimensions. The figure shows that our im2win-based convolution algorithm dominantly uses less memory footprint over all twelve benchmarks compared with the im2col-based convolution in cuBLAS and the fastest convolution among the six algorithms in cuDNN. On average, our algorithm uses 23.1% less memory than cuBLAS, and uses 32.8% less memory than cuDNN. Our algorithm has slightly higher memory usage than

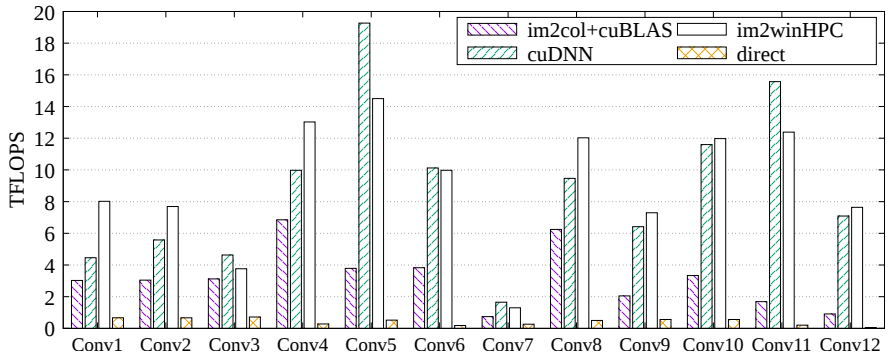

Fig. 2: Performance comparison of our im2win-based convolution with the direct convolution, the im2col-based convolution using cuBLAS and the convolutions in cuDNN (see Table 3).

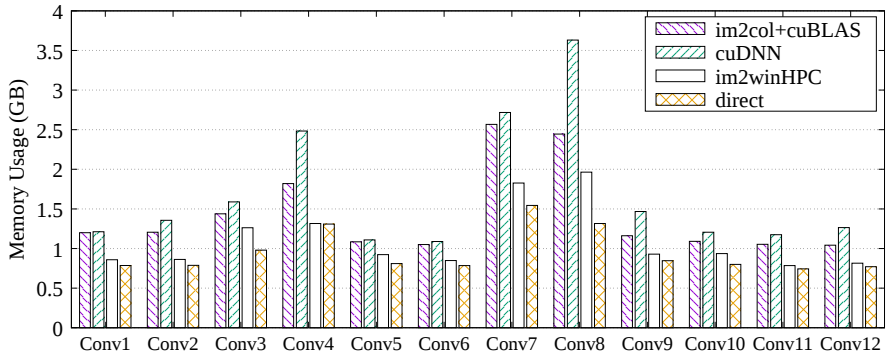

Fig. 3: Memory usages of our convolution compared to the direct convolution as well as PyTorch's im2col+cuBLAS convolution and cuDNN convolutions.

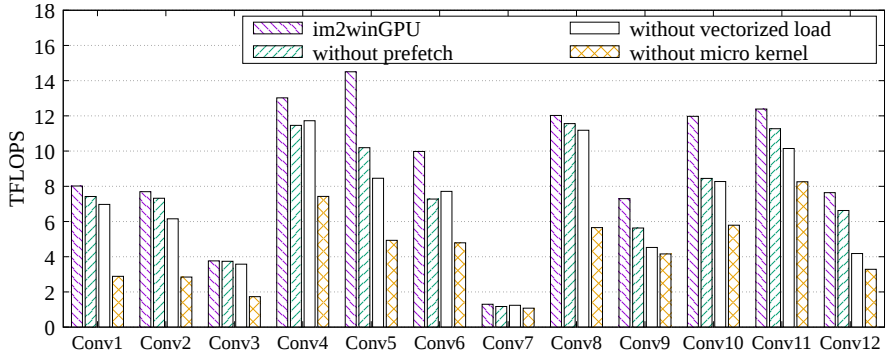

Fig. 4: Performance comparison of the ablation study on the prefetching, the vectorized load, and the micro-kernel optimization techniques. One technique is removed at a time.

the direct convolution. Considering that the memory of a single GPU is usually not big (even Nvidia A100 has at most 80GB of memory), our convolution paradigm supports substantially larger tensor to be processed on a single GPU over cuBLAS and cuDNN, which is much preferable.

4.4 Ablation Study

To explore the performance impact of the prefetching (along with double buffer), the vectorized load, and the micro-kernel techniques we apply in our kernel, we conduct an ablation study on our high-performance `im2win`-based convolution paradigm. We have im2winGPU as the baseline, which includes all the optimization techniques. For other three variants, we remove one technique at a time to study its effectiveness. Figure 4 shows the performance impact of different optimization techniques on our convolution paradigm in terms of the TFLOPS metric. Among the twelve benchmarks, the micro-kernel technique gives the greatest performance boost, followed by the vectorized load, and the prefetching gives the poorest performance boost for our paradigm.

With the micro-kernel implemented as outer product multiplications of vectors in a thread of a block, each thread computes multiple elements of the output tensor O instead of one. This reduces the number of memory accesses and improves the parallelism and computational intensity of the AXPY operations. The vectorized load improve data IO efficiency and memory throughput by loading or storing multiple contiguous data elements from the shared memory into the register. Allocating two buffers (one for prefetching, the other for computation) cuts the size of the available shared memory and registers during computation by half, resulting minimal performance improvement.

5 Conclusion

In this paper, we proposed a new convolution paradigm on GPU. We implemented a window-order-based convolution (called im2win) on GPU using CUDA along with a range of optimizations, including shared memory, tiling, micro-kernel, double buffer, and prefetching. Using twelve DNN benchmarks, we compared our algorithm with the direct convolution, PyTorch's GEMM-based convolution implementation in cuBLAS and six convolution algorithms in cuDNN. The experimental results demonstrate the superior memory and performance efficiency of our im2win-based convolution paradigm compared with the direct convolution and the im2col-based convolution and show comparable performance with the implicit GEMM-based convolution, the FFT convolution, and the Winograd convolution in cuDNN with much less memory footprint.

Acknowledgment This work was partly supported by National Natural Science Foundation of China (Grant No. 62162045), Key Research 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. cuBLAS library (2023), https://docs.nvidia.com/cuda/cublas/
2. PyTorch (2023), https://github.com/pytorch/pytorch
3. Abadi, M., Barham, P., Chen, J., et al.: Tensorflow: A system for large-scale machine learning. In: Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation. p. 265–283. OSDI'16, USENIX Association, USA (2016)
4. Chellapilla, K., Puri, S., Simard, P.: High performance convolutional neural networks for document processing. In: 10th Int'l Workshop on Frontiers in Handwriting Recog. (2006)
5. Chetlur, S., Woolley, C., Vandermersch, P., Cohen, J., Tran, J., Catanzaro, B., Shelhamer, E.: cuDNN: Efficient primitives for deep learning. CoRR **abs/1410.0759** (2014)
6. Cho, M., Brand, D.: MEC: memory-efficient convolution for deep neural network. In: International Conference on Machine Learning (ICML). pp. 815–824. PMLR (2017)
7. Crowley, E.J., Gray, G., Storkey, A.J.: Moonshine: Distilling with cheap convolutions. Advances in Neural Information Processing Systems **31** (2018)
8. Dongarra, J., Hammarling, S., Higham, N.J., Relton, S.D., Valero-Lara, P., Zounon, M.: The design and performance of batched BLAS on modern high-performance computing systems. Procedia Computer Science **108**, 495–504 (2017)
9. Gunnels, J.A., Henry, G.M., Geijn, R.A.: A family of high-performance matrix multiplication algorithms. In: International Conference on Computational Science. pp. 51–60 (2001)
10. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: IEEE Conference on Computer Vision and Pattern Recognition (CVPR). pp. 770–778 (2016)
11. Huang, J., Yu, C.D., van de Geijn, R.A.: Implementing strassen's algorithm with CUTLASS on NVIDIA volta gpus. CoRR **abs/1808.07984** (2018)
12. Jordà, M., Valero-Lara, P., Peña, A.J.: Performance evaluation of cuDNN convolution algorithms on NVIDIA Volta GPUs. IEEE Access **7**, 70461–70473 (2019)
13. Lavin, A., Gray, S.: Fast algorithms for convolutional neural networks. In: IEEE Conference on Computer Vision and Pattern Recognition (CVPR). pp. 4013–4021 (2016)
14. Lu, S., Chu, J., Liu, X.T.: Im2win: Memory efficient convolution on SIMD architectures. In: 2022 IEEE High Performance Extreme Computing Conference (HPEC). pp. 1–7 (2022)
15. Ma, N., Zhang, X., Zheng, H.T., Sun, J.: Shufflenet v2: Practical guidelines for efficient cnn architecture design. In: Proceedings of the European conference on computer vision (ECCV). pp. 116–131 (2018)
16. Oyama, Y., Ben-Nun, T., Hoefler, T., Matsuoka, S.: Accelerating deep learning frameworks with micro-batches. In: 2018 IEEE International Conference on Cluster Computing (CLUSTER). pp. 402–412 (2018)
17. Paszke, A., Gross, S., Massa, F., et al.: Pytorch: An imperative style, high-performance deep learning library. In: Proceedings of the 33rd International Conference on Neural Information Processing Systems (NeurIPS). pp. 8024–8035 (2019)
18. Simonyan, K., Zisserman, A.: Very deep convolutional networks for large-scale image recognition. CoRR **abs/1409.1556** (2015)
19. Song, Z., Wang, J., Li, T., Jiang, L., Ke, J., Liang, X., Jing, N.: Gpnpu: enabling efficient hardware-based direct convolution with multi-precision support in gpu tensor cores. In: 2020 57th ACM/IEEE Design Automation Conference (DAC). pp. 1–6. IEEE (2020)
20. Sze, V., Chen, Y.H., Yang, T.J., Emer, J.S.: Efficient processing of deep neural networks: A tutorial and survey. Proceedings of the IEEE **105**(12), 2295–2329 (2017)
21. Wolfe, M.: Iteration space tiling for memory hierarchies. In: Proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing. p. 357–361. SIAM, USA (1989)
22. Zhang, J., Franchetti, F., Low, T.M.: High performance zero-memory overhead direct convolutions. In: International Conference on Machine Learning. pp. 5776–5785. PMLR (2018)