Enabling Memory-efficient Im2win Convolution with Multi-precision Support on GPU CUDA and Tensor Cores

Xiang Fu* Jixiang Ma* Xinpeng Zhang* Peng Zhao† Shuai Lu* Xu T. Liu‡§
*Nanchang Hangkong University, China †EEO Tech, China ‡University of Washington, USA §AWS, USA

***Abstract*—Convolution is a principal computational bottleneck in deep neural networks, and its efficiency depends on tight integration between algorithms and GPU hardware. Existing GPU convolution methods suffer from large memory overhead, poor cache utilization, limited effectiveness across kernel sizes, or numerical instability. This work extends the im2win paradigm—a universal, memory-efficient convolution method with contiguous memory access for all kernel sizes—to run efficiently in full precision on CUDA cores and half precision on tensor cores. By introducing new kernel designs and optimizations such as zig-zag memory access and asynchronous data movement, im2win efficiently exploits hardware-accelerated half-precision matrix multiply-accumulate operations. Across twelve CNN benchmarks, im2win achieves up to 2.8× higher TFLOPS than its CUDA core implementation, 1.4× higher than cuDNN, and 6.4× higher than GEMM-based convolution with cuBLAS, while using as little as 53% and 35% of their memory, respectively. These results establish im2win as a unified, high-performance convolution framework for modern GPU architectures.**

I. Introduction

Convolutional Neural Networks (CNNs), foundational to deep learning for spatial tasks like computer vision, rely heavily on convolution operations that account for 50–90% of execution time [1]. Efficient GPU convolution demands leveraging CUDA cores for full-precision processing and tensor cores—introduced in NVIDIA's Volta architecture—for mixed-precision acceleration, to maximize performance and minimize memory usage.

To enhance efficiency across diverse hardware, there are convolution algorithms like direct convolution which has zero memory overhead but poor cache utilization [2], FFT (Fast Fourier Transform) convolution which has speedups for large kernels but latency on small filters, and Winograd convolution which has computation reduction for small fixed kernels but numerical instability at larger sizes. Im2col-based GEMM methods leverage libraries like cuBLAS but suffer high memory overhead from data transformation [3] and irregular matrices, yielding suboptimal performance [4]. GPU memory constraints drive processing strategies—from PyTorch's single-batch to cuDNN's mini-batch—and Implicit GEMM on tensor cores for efficient mixed-precision convolution without explicit intermediates.

Unlike prior methods limited by memory overhead (im2col), poor cache utilization (direct), latency on small kernels (FFT), or numerical instability (Winograd), the im2win (image to window) convolution paradigm addresses the high memory overhead and poor locality issues in im2col-based and direct convolutions [5], [6]. It achieves this by transforming the input tensor into an im2win tensor that enables contiguous memory access and data reuse, significantly reducing memory consumption compared to im2col.

Inspired by im2win, we extend it for GPU architectures with multi-precision support, implementing full-precision kernels on CUDA cores and half-precision kernels on tensor cores. To maximize hardware utilization on tensor cores, we apply additional kernel designs and optimization techniques such as index precomputation, zig-zag memory access and asynchronous data movement. With thorough evaluation on twelve state-of-the-art CNN benchmarks, we compare our im2win convolution against PyTorch's GEMM-based convolution using cuBLAS and multiple cuDNN convolution variants. We further conduct an ablation study to evaluate the performance impact of individual optimization techniques.

To summarize, this paper makes the following **main contributions** and is organized as follows:

1) We extend the im2win convolution paradigm to support multi-precision execution, implementing full-precision kernels for CUDA cores and half-precision kernels for tensor cores in Sec. III.

2) We augment the im2win convolution on CUDA and tensor cores, along with a suite of optimizations in Sec. III-E. We analyze the effect of core optimizations through an ablation study in Sec. IV-E.

3) We conduct a comprehensive experimental evaluation comparing optimized im2win convolution, cuDNN algorithms and PyTorch's im2col-based convolution implementations across multiple precisions and on both CUDA and tensor cores in Sec. IV. [1]

[1]Code - https://github.com/TensorConv/im2win/tree/IJCNN_2026

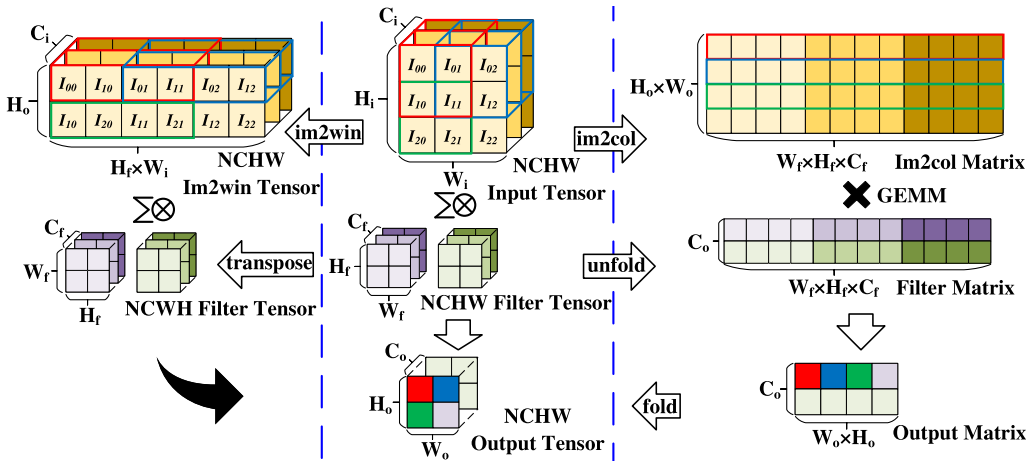

Figure 1. The schematic diagram illustrates, from left to right, direct convolution, im2win convolution, and im2col-based convolution, separated by blue dashed lines. From top to bottom, these represent the input (1×3×3×3×3), filter (2×3×2×2), and output (1×2×2×2). Colors denote distinct channels, while arrows indicate transformation processes. Red, blue, and green boxes within the input highlight the input regions required to compute the first three matching output elements. These regions are multiplied by the filter and summed to yield the corresponding output elements.

II. Related Work

Convolution optimization strategies diverge significantly based on the underlying hardware architecture. On CPUs, implementations typically rely on either optimized direct convolution, employing micro-kernel reordering and cache-friendly layouts to enhance locality [7], or explicit GEMM-based methods (im2col), which transform convolution into matrix multiplication [8] and may split large matrices to manage memory overhead [9]. In contrast, on GPUs, to overcome the bandwidth bottlenecks of explicit data expansion, implicit GEMM strategies are widely adopted to compute indices on-the-fly [10]; furthermore, recent advancements leverage specialized hardware like Tensor Cores, to accelerate these operations with arbitrary precision support [11].

III. Im2win

To address the large memory usage of the im2col-based convolution and the inefficient, non-consecutive memory access of direct convolution, the im2win paradigm was proposed [5], [6]. Im2win dramatically reduces memory overhead through a more compact data arrangement than im2col, enabling consecutive memory access and improved data reuse. This section describes the im2win paradigm, including its data transformation and convolution steps, and how we implement a high-performance im2win algorithm on tensor cores.

A. Notation

The convolution operation includes an input tensor ($\mathcal{I}$), a filter tensor ($\mathcal{F}$), and an output tensor ($\mathcal{O}$). In these tensors, N is the batch size, s is the stride size. Let $C_{i/f/o}$, $H_{i/f/o}$ and $W_{i/f/o}$ denote the channel, height and width of tensors respectively. The input, filter and output tensors in NCHW data layout are denoted as $\mathcal{I}[N_i][C_i][H_i][W_i]$, $\mathcal{F}[N_f][C_f][H_f][W_f]$ and $\mathcal{O}[N_o][C_o][H_o][W_o]$ respectively. The (2D) convolution is defined as:

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

where $1 \le i \le N_o, 1 \le j \le C_o, 1 \le m \le H_o, 1 \le n \le W_o, 1 \le r \le C_f, 1 \le u \le H_f, 1 \le v \le W_f$.

Tensors are often stored as one-dimensional arrays in memory, although logically represented as four-dimensional arrays, the cost of accessing different dimensions varies with different data layouts [12]. In the NCHW layout, the elements in the width dimension are contiguous in memory by prioritizing the width dimension first which means the cost of contiguous accessing the width dimension is the lowest, followed by height, channel and batch. NCHW and NHWC are more popular on CPUs [13], while NCHW and CHWN are used on GPUs [14].

B. Im2win Data Transformation

As shown in the red/blue/green boxes in Fig. 1, these input elements used to compute an output element are considered as a sliding window, The im2win transform enables the elements within these sliding windows to be arranged consecutively across different channels. The red boxes (in three channels) are the first window whose elements are used to compute the first element of the output tensor, the blue boxes the second, and the green boxes the third. The im2win transformation is as follows:

$$\hat{\mathcal{I}}(i, r, m, k \times H_f + u) = \mathcal{I}(i, r, m \times s + u, k), \quad (2)$$

where $1 \le i \le N_o, 1 \le r \le C_o, 1 \le m \le H_o, 1 \le k \le W_i, 1 \le u \le H_f$.

The upper right of Fig. 1 illustrates the im2col transformation, where each row of the resulting matrix contains the elements of a sliding window. This approach leads to significant redundancy, as many elements are duplicated across neighboring rows, resulting in a much larger memory usage than the original input tensor. Compared to the im2col matrix, $\hat{I}$ saves memory by eliminating redundant storage of overlapping elements. Specifically, it reduces the required space by $N_o \times C_o \times H_o \times W_o \times C_i \times H_f \times (W_f - s)$. Notably, when the stride s equals the filter height H_f, the space used by the im2win tensor $\hat{I}$ matches that of the im2col matrix, since there is no element reuse. However, as long as $s < H_f$, $\hat{I}$ requires less memory than the im2col matrix.

C. Im2win Convolution

A naive im2win convolution is illustrated in [5], which uses a seven-layer nested loop structure like direct convolution. In Fig. 1, direct, im2win, and im2col convolutions are shown with their respective input formats at the top of the figure. For each output element, the algorithm multiplies the elements in a window (red box) with the corresponding filter elements and sums the results. The second output element takes the elements within the blue box as its input, representing the subsequent window. In im2win convolution, the window shifts by $s \times H_f$ horizontally; for advancing to the next column of the output, the window moves down by 1. For each output channel C_o, the corresponding filter is selected.

Both direct and im2win convolutions compute all output elements by iterating through the outer four loops, with the innermost loops handling the detailed element-wise multiplications. The operation can be formally described as:

$$\hat{I}_{(i,r,m,(n\times s)\times W_f + v \times W_f + u)} \times \mathcal{F}_{(j,r,u,v)}, \qquad (3)$$

where the indices correspond to batch, channel, spatial, and filter dimensions.

The time complexity of im2win convolution is $O(N_o \times C_o \times H_o \times W_o \times C_f \times H_f \times W_f)$, which matches that of direct convolution. For filter data continuity, im2win leverages a filter tensor layout (such as NCHW) that prioritizes elements along the height before the width, aligning with the im2win access pattern. Im2win further optimizes performance by reusing elements between overlapping windows in the same column, allowing elements loaded in a previous window to remain in cache for subsequent computations. This reduces memory loading time and improves overall efficiency compared to both direct and im2col-based convolutions.

Algorithm 1: High-Performance Im2win Convolution on Tensor Cores

Input: im2win input tensor $\hat{I}$, filter tensor $\mathcal{F}$, stride s
Output: output tensor O

Dimensions: $\mathbf{M} = C_o$, $\mathbf{N} = N_o \times H_o \times W_o$, $\mathbf{K} = C_f \times H_f \times W_f$
# of blocks: $M/M_B \times N/N_B$
 # number of threads : T
 # warp per block: $M_B/M_W \times N_B/N_W$
WMMA Registers: $A_{reg}[2][N_W], B_{reg}[2][M_W], C_{reg}[M_W][N_W]$
Shared memory: $A_{shm}[2][K_W \times N_B], B_{shm}[2][K_W \times M_B]$
$bx \leftarrow (blockIdx.z \bmod 2 = 0)\ ?\ blockIdx.x\ :\ (gridDim.x - blockIdx.x)$
$by \leftarrow blockIdx.y + blockIdx.z \times gridDim.y$
$A_{shm}[0][K_W \times N_B/T] \overleftarrow{async\_vec\_load}(\hat{I}(by, 0))$
$B_{shm}[0][K_W \times M_B/T] \overleftarrow{async\_vec\_load}(\mathcal{F}(bx, 0))$
commit_and_wait()
wmma::load_matrix_sync($A_{reg}[0]$,$A_{shm}[0][K_W \times N_B]$)
wmma::load_matrix_sync($B_{reg}[0]$,$B_{shm}[0][K_W \times M_B]$)
for $k = 1$ **to** K/K_W **do**
  wmma::mma_sync(C_{reg}, $B_{reg}[write]$, $A_{reg}[write]$, C_{reg})
  if $k \neq K/K_W$ **then**
    $A_{shm}[read][K_W \times N_B/T] \overleftarrow{async\_vec\_load}(\hat{I}(by, k))$
    $B_{shm}[read][K_W \times M_B/T] \overleftarrow{async\_vec\_load}(\mathcal{F}(bx, k))$
    commit_and_wait()
    wmma::load_matrix_sync($A_{reg}[read]$, $A_{shm}[read][K_W \times N_B]$)
    wmma::load_matrix_sync($B_{reg}[read]$, $B_{shm}[read][K_W \times M_B]$)
  __syncthreads()
$O(bx, by) \overleftarrow{write}(C_{reg})$

D. Im2win Convolution on Tensor Cores

We implement the im2win convolution on tensor cores using the NHWC layout to enhance memory access contiguity for convolutional windows [12]. The design aligns with the architectural features of CUDA and tensor cores, with careful consideration of thread-block organization, shared memory allocation, and warp-level computation. Shared memory is managed at the thread-block level, where each block contains multiple warps and threads. Data loading occurs at the thread level, while matrix multiplications on tensor cores are executed at the warp level.

We proposed high-performance im2win convolution algorithm implemented on tensor cores is presented in Algo. 1. The algorithm begins by tiling the output tensor. Since the im2win convolution requires low extra memory, the im2win convolution can process all batches simultaneously to achieve higher parallelism. We map the output indices into two logical

dimensions: $\mathbf{M} = C_o$ and $\mathbf{N} = N_o \times H_o \times W_o$. Each thread block is assigned a micro-tile of size $M_B \times N_B$. As we use tensor cores via warp matrix-multiply accumulate (WMMA) instructions with 16×16 granularity, both M_B and N_B are constrained to be multiples of 16. While larger blocks improve tensor cores utilization, they reduce the number of concurrent thread blocks and increase shared memory pressure. To strike a balance, we adopt two block shapes: 32×128 and 64×128.

For dimension N, we further decompose the mapping into two dimensions and traverse dimension N in a zig-zag pattern by alternating parity order, thereby reducing memory bank conflicts and irregular access strides shown in Ln. 7. Within each thread block, computations are further distributed across multiple warps, with each warp responsible for an $M_W \times N_W$ tile. This ensures sufficient computation per warp to fully utilize tensor cores while maintaining adequate parallelism across warps.

Data in convolutional windows are loaded in a tiled manner. Input data required for convolution are fetched from global memory into shared memory using vectorized 16-byte load operations, with asynchronous data movement implemented through Parallel Thread Execution (PTX) instructions in Lns. 9 and 10. The data are moved from shared memory to registers using vectorized load operations in Lns. 12 and 13. To hide the data movement latency, we allocate two shared memory buffers and two registers in Lns. 5 and 6: one contains the data in current computation while the other pre-fetches for the next computation in Lns. 17 and 18. Upon computation, the results are written back to the output tensor.

E. Optimizations for im2win convolutions on CUDA and Tensor Cores

This subsection outlines optimizations for im2win convolutions, designed to fully harness the potential of tensor cores while maximizing computational throughput and memory bandwidth. We employed a suite of common optimization techniques, including: tiling and data pre-fetching, shared memory and register usage, vectorized load/store operations, double buffering. We further leveraged the capabilities of the NVIDIA SM 80 architecture by introducing two additional methods: asynchronous data movement and Zig-Zag access. Owing to space constraints, common optimization techniques shall not be elaborated upon here; their specific implementation details may be found in the relevant literature [6].

1) index precomputation: As the index offsets required for each thread in the convolution are fixed, these offsets can be precomputed on the host and stored in GPU constant memory. This approach is primarily applied during the process of loading input tensors from global memory into shared memory, thereby avoiding redundant index calculations on the device and effectively reducing runtime overhead.

2) Asynchronous data movement: The main purpose of asynchronous data movement is to maximize computation utilization by separating the two tasks, data movement and data computation, so that they can be performed concurrently, thus hiding the data movement latency. We implement the underlying asynchronous data movement by using the PTX instruction. To further increase the utilization of the compute units and completely hide the data movement latency, we usually combine the asynchronous data movement with double buffering to build an efficient producer-consumer pipeline.

3) Zig-Zag access: In convolutional computation, when the thread block division of a certain dimension is too large, it easily leads to bank conflicts during shared memory access, which reduces the memory access efficiency. To alleviate this problem, the Zig-Zag access pattern can be used to rearrange the data access order [15]. We introduce an additional access dimension into the thread block division, and decide the access direction according to the parity of this dimension. This alternate access strategy significantly reduces the bank conflict caused by multiple thread blocks accessing the same address at the same time, and improves the throughput of parallel access.

F. Multi-precision Support

Convolution traditionally relies on single precision (FP32), half precision (FP16) offers superior memory bandwidth. and deliver higher GFLOPS rates [16]. Our implementation supports two optimized variants: an FP32 version on CUDA cores and an FP16 version on tensor cores. On Ampere-based GPUs—the platform for our experiments—performing FP16 arithmetic without tensor cores often requires internal casting to FP32, which reduces performance benefits. Since Tensor Cores natively support efficient FP16 matrix multiply-accumulate (MMA), our FP16 variant leverages tensor cores using the WMMA API to achieve higher throughput.

IV. Experiments

This section outlines the experimental setup, detailing the hardware configuration, benchmarks, and algorithms used for comparative analysis. Subsequently, it presents a comprehensive analysis of the GPU-based experimental results, focusing on key performance metrics such as TFLOPS and MaxRSS for each algorithm.

A. Experimental Setup

1) Experiment environment: All experiments were conducted on a system equipped with an NVIDIA® GeForce RTX® 3090 GPU (24 GB memory, Ampere architecture featuring third-generation tensor cores),

hosted on an Intel® Xeon® Silver 4214 CPU server. The implementation was built with CUDA 11.3 and integrated with PyTorch 2.2.0. For baseline comparisons, we evaluated against PyTorch's GEMM-based convolution backend (linked to cuBLAS 11.3) as well as the convolution primitives in cuDNN 8.2.1. Peak GPU memory consumption was monitored in real time using the NVIDIA System Management Interface (nvidia-smi).

2) cuDNN convolutions: On CUDA cores, we study seven convolution implementations in cuDNN, which are grouped into im2col-based (im2col), implicit GEMM-based (IPG), FFT (FT), and Winograd (WN) approaches. Due to space limit, we present the results of the optimized variants of IPG, FT, and WN, omitting their less optimized ones. cuDNN dynamically selects the most efficient algorithm at runtime based on hardware characteristics, stride, input and filter tensor properties. Consequently, the selected algorithm may vary across different runs. When leveraging tensor cores, cuDNN utilizes only the IPG and WN implementations.

3) im2win convolutions: We develop two variants of the im2win convolution: an FP32 implementation on CUDA cores and an FP16 one on tensor cores. The CUDA-core version builds upon prior work [6], enhanced with two new optimizations: index precomputation and vectorized data transfers between global and shared memory. We conduct an ablation study (see Sec. IV-E) with four implementations on tensor cores to evaluate the performance impact of individual one: a version incorporates all optimizations except index precomputation outlined in Sec. III-E, one without Zig-Zag access (wo_zigzag), one without asynchronous data movement (wo_async), and one without double buffering (wo_double).

4) im2col convolutions with cuBLAS: We compare with PyTorch's implementation, backed by cuBLAS for the GEMM computation. We evaluate both FP32 on CUDA cores and FP16 on tensor cores.

In all figures and tables in this section, "CC" denotes results obtained using FP32 precision executed on CUDA cores, while "TC" denotes results obtained using FP16 precision executed on tensor cores.

B. Benchmarks

To adequately cover the diversity of convolutional layers present in modern deep neural networks (DNNs), evaluating a single model with uniform filter dimensions—such as VGG-16, which exclusively employs 3×3 kernels, or ResNet-50, which utilizes only three distinct kernel sizes—is insufficient. To address this limitation, our evaluation adopts a state-of-the-art DNN benchmark that includes twelve unique convolutional benchmarks (cv1–cv12), detailed in Tab. I.

Table I
TWELVE CONVOLUTIONAL 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 $C_o \times H_o \times W_o$
cv1	$3 \times 227 \times 227$	$96 \times 11 \times 11, 4$	$96 \times 55 \times 55$
cv2	$3 \times 231 \times 231$	$96 \times 11 \times 11, 4$	$96 \times 56 \times 56$
cv3	$3 \times 227 \times 227$	$64 \times 7 \times 7, 2$	$64 \times 111 \times 111$
cv4	$64 \times 224 \times 224$	$64 \times 7 \times 7, 2$	$64 \times 109 \times 109$
cv5	$96 \times 24 \times 24$	$256 \times 5 \times 5, 1$	$256 \times 20 \times 20$
cv6	$256 \times 12 \times 12$	$512 \times 3 \times 3, 1$	$512 \times 10 \times 10$
cv7	$3 \times 224 \times 224$	$64 \times 3 \times 3, 1$	$64 \times 222 \times 222$
cv8	$64 \times 112 \times 112$	$128 \times 3 \times 3, 1$	$128 \times 110 \times 110$
cv9	$64 \times 56 \times 56$	$64 \times 3 \times 3, 1$	$64 \times 54 \times 54$
cv10	$128 \times 28 \times 28$	$128 \times 3 \times 3, 1$	$128 \times 26 \times 26$
cv11	$256 \times 14 \times 14$	$256 \times 3 \times 3, 1$	$256 \times 12 \times 12$
cv12	$512 \times 7 \times 7$	$512 \times 3 \times 3, 1$	$512 \times 5 \times 5$

Table II
THE OBSERVED BEST PERFORMANT CUDNN CONVOLUTIONS AND IM2WIN IN EACH BENCHMARK.

NAME	CUDNN ALGO CUDA CORES	CUDNN ALGO TENSOR CORES	IM2WIN ALGO TENSOR CORES
cv1	IPG_CC	IPG_TC	baseline
cv2	IPG_CC	IPG_TC	wo_zigzag
cv3	IPG_CC	IPG_TC	baseline
cv4	FT_CC	IPG_TC	wo_async
cv5	WN_CC	IPG_TC	baseline
cv6	WN_CC	WN_TC	baseline
cv7	IPG_CC	IPG_TC	wo_double
cv8	FT_CC	WN_TC	baseline
cv9	FT_CC	IPG_TC	baseline
cv10	FT_CC	WN_TC	baseline
cv11	WN_CC	WN_TC	wo_async
cv12	IPG_CC	WN_TC	baseline

C. Overall Performance on CUDA and Tensor Cores

We compare our im2win convolution with PyTorch's im2col-based convolution with cuBLAS and the cuDNN convolutions. We set N_i=256 and run in each of the twelve convolutional benchmarks 50 times, reporting the highest observed TFLOPS and lowest MaxRSS. Among four im2win_TC versions, we report the version with the best performance for each benchmark, shown as the last column in Tab. II. Similarly, the auto-selected algorithm in cuDNN on CUDA and tensor cores are reported in the second and third columns respectively in Tab. II.

Fig. 2 summarizes the TFLOPS and memory usage of each method. Im2win_TC achieves a 2.7× increase in overall TFLOPS compared to im2win_CC. This performance gain arises from two main factors. First, im2win_TC employs the FP16 data type, whereas im2win_CC uses FP32, allowing im2win_TC to process twice as many elements for the same data transfer volume. Second, tensor cores are optimized for matrix operations and can execute block-level computations in a single instruction, while CUDA cores perform scalar operations and handle one data element per instruction.

Im2win_TC achieves a 1.4× improvement in overall TFLOPS compared to cuDNN_TC and 6.4× compared to im2col_cuBLAS_TC. The underlying

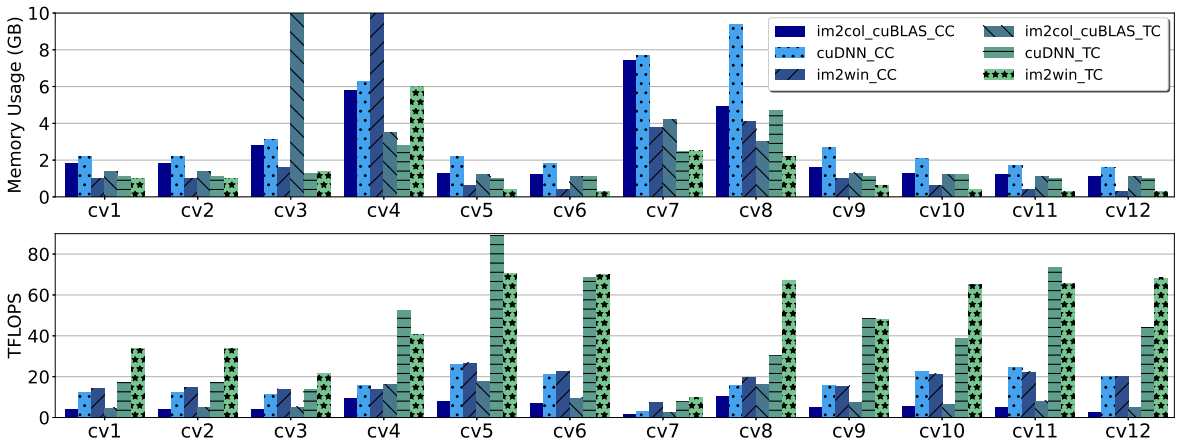

Figure 2. Performance results in TFLOPS and memory usage among the three convolutional algorithms (im2win, im2col_cuBLAS, and cuDNN) on CUDA and tensor cores respectively. Note: that the memory usage of im2col_cuBLAS_TC is 14.6 GB on cv3, and that of im2win_CC is 11.7 GB on cv4.

reasons for these performance gains will be examined in detail in the following subsection.

In terms of memory consumption, im2win_TC uses on average about 53% as much memory as cuDNN_TC and approximately 35% as much as im2col_cuBLAS_TC. Im2win_CC consumes less memory than cuDNN_CC across all benchmarks except cv4, while im2win_TC consumes less memory than cuDNN_TC across all benchmarks except cv3 and cv4. The specific causes of this behavior will be analyzed in detail in the following subsection.

D. Micro Benchmark with Each Convolution

To explicitly compare our im2win implementations with each cuDNN convolution variant, we modify PyTorch's convolution implementation to override the default adaptive algorithm selection and enforce a specific cuDNN algorithm for each benchmark. However, due to algorithmic constraints, certain algorithms may not be executable on specific layers. For details, please consult the cuDNN documentation. We present the performance results in TFLOPS and the memory usage of the im2win, PyTorch's im2col-based convolutions using cuBLAS, and cuDNN convolutions on GPU CUDA and tensor cores in Fig. 3.

On CUDA cores, the im2win algorithm achieves the highest TFLOPS on six convolutional benchmarks, while the cuDNN_IPG implementation performs best on the remaining six benchmarks. For memory, im2win minimizes usage across all cases except cv4, which slightly exceeds cuBLAS_im2col.

On Tensor Cores, im2win delivers peak TFLOPS on eight benchmarks; cuDNN_IPG leads on cv3 and cv4, while cuDNN_WN dominates cv11.

1) Im2win vs cuBLAS_im2col vs cuDNN_im2col: Overall, im2win_CC delivers a 3.4× speedup over cuBLAS_im2col_CC and a 2.7× speedup over cuDNN_im2col_CC. Likewise, im2win_TC achieves a 6.4× speedup compared to cuBLAS_im2col_TC.·

In terms of memory consumption, im2win_CC achieves the lowest consumption across all benchmarks except cv4, on average about 62% as much memory as cuBLAS_im2col_CC, about 41% as much memory as cuDNN_im2col_CC. For im2win_TC, it continues to maintain the smallest memory requirement on all remaining benchmarks except cv3, cv4 and cv8, on average about 55% as much memory as cuBLAS_im2col_TC.

The superior efficiency of im2win arises from its data transformation. Unlike the im2col approach, im2win flattens the convolution window, which reduces redundant data storage and enhances element reuse and spatial locality, as illustrated in Fig. 1.

Recall there are three processing strategies when processing tensors in convolution: single-batch, mini-batch, or full-batch. The cuBLAS_im2col implementation in PyTorch adopts a single-batch strategy, which allocates additional memory for an intermediate tensor corresponding to each batch. In contrast, im2win processes all batches concurrently, resulting in higher memory usage for certain cases such as cv4. Meanwhile, cuDNN_im2col utilizes a mini-batch strategy, requiring more intermediate storage than the single-batch approach of cuBLAS_im2col but less than full-batch processing.

2) Im2win vs cuDNN_IPG: Compared to cuDNN's implicit GEMM-based approach, im2win delivers comparable TFLOPS performance on CUDA cores. While on tensor cores, im2win achieves an overall 2× performance improvement over cuDNN. This performance gain primarily stems from the optimizations introduced in Sec. IV-C, namely Zig-Zag access and asynchronous data movement.

Since cuDNN adopts a mini-batch strategy whereas

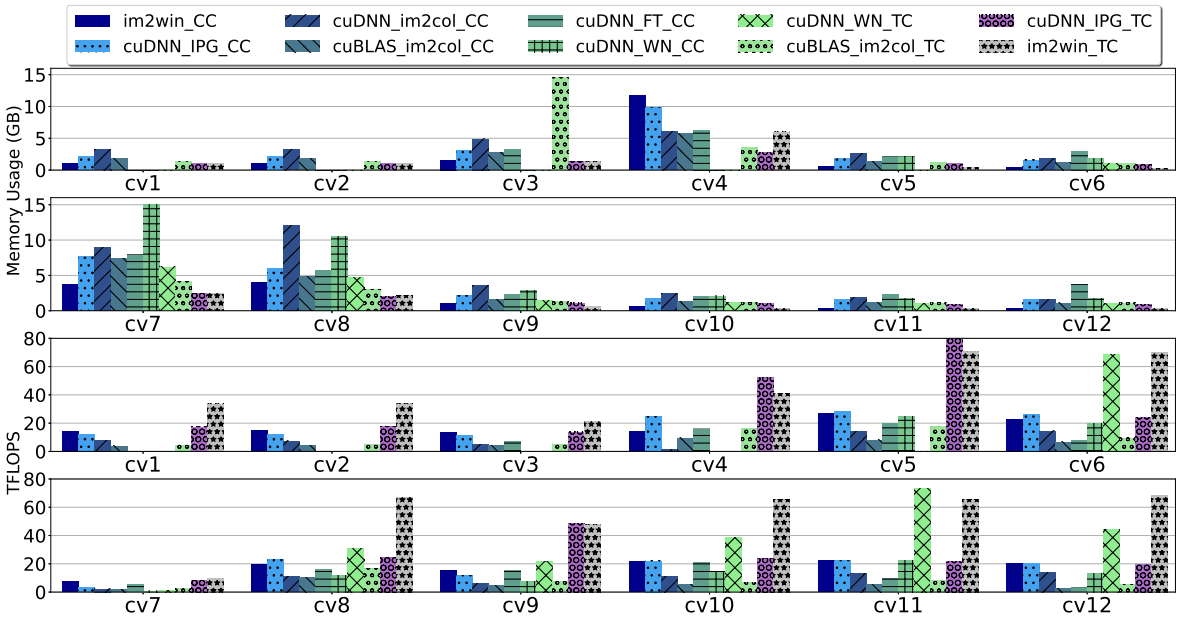

Figure 3. Performance results in TFLOPS and the memory usage of the im2win, PyTorch's im2col-based convolutions using cuBLAS, and cuDNN convolutions on GPU CUDA cores and tensor cores. Notes that some of these algorithms in cuDNN fail to run on some benchmarks due to certain constraints and TFLOPS of cuDNN_IPG_TC is 88.86 on cv5.

im2win employs a full-batch approach for tensor processing, im2win incurs higher memory consumption than cuDNN's implicit GEMM-based method in certain benchmarks. Nevertheless, on CUDA cores, im2win consistently consumes less memory than IPG across all benchmarks, averaging approximately 46% of its memory usage. Excluding benchmarks cv3, cv4, and cv8, im2win_TC further reduces memory consumption to an average of 60% of cuDNN_TC.

3) Im2win vs cuDNN_WN: Compared to cuDNN_WN_CC, im2win_CC achieves 2.5× higher overall TFLOPS, while im2win_TC attains 2.8× overall TFLOPS of cuDNN_WN_TC. In terms of memory usage, im2win demonstrates substantially lower requirements; im2win_CC consumes, on average, 27% of the memory used by cuDNN_WN_CC, and im2win_TC averages 37% of the memory usage of cuDNN_WN_TC. The Winograd algorithm significantly reduces the number of multiplications by introducing additional addition operations via linear transformation (see Sec. II for details). While this approach surpasses im2win in computationally small benchmarks, the associated transformations incur extra memory overhead, giving im2win a clear advantage in memory efficiency.

4) Im2win vs cuDNN_FT: Compared to the FFT-based approach im2win achieves 2.0× higher overall TFLOPS. In terms of memory consumptions, im2win requires on average 49% of the memory required by cuDNN_FT_CC.

The FFT-based algorithm performs a transformation of the tensor from time to frequency domain (see Sec. II for details), which is the main computational overhead. This overhead becomes more pronounced for larger tensor sizes, explaining why the FFT algorithm outperforms im2win on benchmark cv4. However, storing the intermediate tensor generated during the transformation introduces additional memory requirements, giving im2win lower overall memory overhead compared to the FFT-based approach.

E. Ablation Study on Optimization Techniques

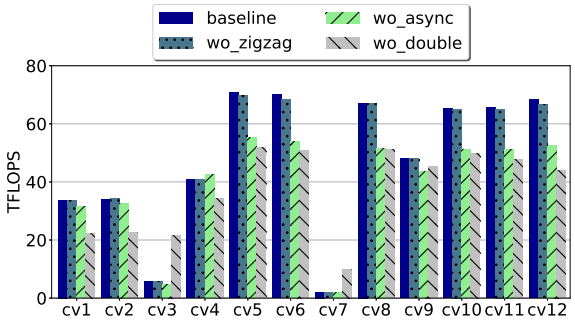

Figure 4. Performance results in TFLOPS of the ablation study of three optimization techniques on tensor cores.

To evaluate the individual impact of three optimization techniques, namely asynchronous data movement, double buffering and Zig-Zag access, we conducted an ablation study on the im2win convolution on tensor cores. The baseline includes all the proposed optimizations while three other variants disable one optimization at a time to evaluate its independent contribution. Fig. 4 shows the performance results

in TFLOPS across four optimization configurations. The results indicate that double buffering generally delivers the greatest performance gains, followed by asynchronous data movement, whereas Zig-Zag access yields the smallest improvement.

When assigning tasks to thread blocks, we typically chunk the convolution window, computing the elements corresponding to only one of the sub-blocks at a time. The computation proceeds in two phases: (i) loading input data from global memory into shared memory, and (ii) reading data from shared memory into registers for computation. Since the computation operation depends on the completion of the corresponding data load, we use the double buffering to implement the transfer of the next part of the data while computing the first part of the data, thus hiding the overhead of partial data transfer.

However, in benchmarks with small convolution windows, the performance is instead improved by removing the double buffering. This is mainly due to the fact that the double buffering itself introduces additional overheads, including latency from synchronization operations and additional share memory usage. When the convolution window is small, its inherent overheads outweigh its gains leading to a negative optimization of the double buffering.

In contrast, asynchronous data movement can also hide some of the data transfer latency, but its effectiveness is typically lower than that of double buffering. The Zig-Zag access pattern, designed primarily to reduce shared memory bank conflicts, delivers the most modest gains. Double buffering and asynchronous data movement have already reduced the probability of bank conflict to a certain extent through the data Pre-Fetching. Consequently, the marginal benefit of Zig-Zag access remains limited.

V. Conclusion

In this paper, we presented an optimized implementation of im2win convolution for modern GPUs, effectively addressing memory overhead and data locality challenges through a novel input tensor transformation. We extended the algorithm to support multi-precision computations, utilizing full-precision on CUDA cores and half-precision on tensor cores. Our kernel optimizations yield significant efficiency gains over standard approaches. Our im2win implementation on CUDA cores shows significance performance improvement over other convolution variants in cuDNN and cuBLAS-based im2col. Finally, our ablation study identifies double buffering as the most critical optimization factor, followed by asynchronous data movement, with zig-zag memory access providing supplementary performance benefits.

Acknowledgment

This research was partially supported by the National Natural Science Foundation of China (Grant No. 42261070). This work is not related to Xu T. Liu's position at Amazon.

References

[1] 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.

[2] A. d. L. Santana, A. Armejach, and M. Casas, "Efficient direct convolution using long simd instructions," in *Proceedings of the 28th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2023, p. 342–353.

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

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

[5] S. Lu, J. Chu, and X. T. Liu, "Im2win: Memory efficient convolution on SIMD architectures," in *2022 IEEE High Performance Extreme Computing Conference (HPEC)*, 2022, pp. 1–7.

[6] S. Lu, J. Chu, L. Guo, and X. T. Liu, "Im2win: An Efficient Convolution Paradigm on GPU," in *Euro-Par 2023: Parallel Processing*, ser. Lecture Notes in Computer Science, Cham, 2023, pp. 592–607.

[7] J. Zhang, F. Franchetti, and T. M. Low, "High performance zero-memory overhead direct convolutions," in *Int'l 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] 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.

[10] M. Jordà, P. Valero-Lara, and A. J. Peña, "Performance evaluation of cuDNN convolution algorithms on NVIDIA Volta GPUs," *IEEE Access*, vol. 7, pp. 70 461–70 473, 2019.

[11] S. Qummar, A. Ernstsson, C. Kessler, and O. Sysoev, "Skepu-dnn: Algorithmic skeleton programming for deep learning on heterogeneous systems," in *2025 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2025, pp. 423–432.

[12] X. Fu, X. Zhang, J. Ma, P. Zhao, S. Lu, and X. T. Liu, "High performance im2win and direct convolutions using three tensor layouts on simd architectures," in *2024 IEEE High Performance Extreme Computing Conference (HPEC)*, 2024, pp. 1–7.

[13] B. Li, Q. Xue, G. Yuan, S. Li, X. Ma, Y. Wang, and X. Tang, "Optimizing data layout for training deep neural networks," in *Companion Proceedings of the Web Conference 2022*, 2022, pp. 548–554.

[14] C. Li, Y. Yang, M. Feng, S. Chakradhar, and H. Zhou, "Optimizing memory efficiency for deep convolutional neural networks on gpus," in *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2016, pp. 633–644.

[15] V. Volkov and J. W. Demmel, "Benchmarking gpus to tune dense linear algebra," in *SC'08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. IEEE, 2008, pp. 1–11.

[16] Z. Song, J. Wang, T. Li, L. Jiang, J. Ke, X. Liang, and N. Jing, "Gpnpu: enabling efficient hardware-based direct convolution with multi-precision support in gpu tensor cores," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2020, pp. 1–6.