Im2col-Winograd: An Efficient and Flexible Fused-Winograd Convolution for NHWC Format on GPUs

Zhiyi Zhang
gilgamesh@mail.ustc.edu.cn
University of Science and Technology of China, Hefei Institutes of Physical Science, Chinese Academy of Science
Hefei, China

Pengfei Zhang
pfzhang@aiofm.ac.cn
Hefei Institutes of Physical Science, Chinese Academy of Sciences
Hefei, China

Zhuopin Xu
xuzp@iim.ac.cn
Hefei Institutes of Physical Science, Chinese Academy of Sciences
Hefei, China

Bingjie Yan
bj.yan@ieee.org
Institute of Computing Technology, Chinese Academy of Sciences
Beijing, China

Qi Wang*
wangqi@ipp.ac.cn
Hefei Institutes of Physical Science, Chinese Academy of Sciences
Hefei, China

ABSTRACT

Compared to standard convolution, Winograd algorithm has lower time complexity and can accelerate the execution of convolutional neural networks. Previous studies have utilized Winograd to implement 2D convolution on GPUs, mainly using 2D Winograd, and arranging tensors in *NCHW* or *CHWN* format instead of *NHWC* to make data access coalesced. Due to the higher space complexity of Winograd and limited hardware resources, these implementations are usually confined to small filters. To provide an efficient and flexible fused-Winograd convolution for *NHWC* format on GPUs, we propose Im2col-Winograd. This algorithm decomposes an ND convolution into a series of 1D convolutions to utilize 1D Winograd, thereby reducing space complexity and data-access discontinuity. The reduced space complexity makes Im2col-Winograd less restricted by hardware capability, enabling it to accommodate a wider range of filter shapes. Our implementations support 2-9 filter widths and do not use any workspace to store intermediate variables. According to the experiments, Im2col-Winograd achieves a speedup of 0.788× to 2.05× over the fastest benchmark algorithm in cuDNN; and shows similar convergence to PyTorch on Cifar10 and ILSVRC2012 datasets. Along with memory efficiency, the more generalized acceleration offered by Im2col-Winograd can be beneficial for extracting features at different convolution scales.

CCS CONCEPTS

• **Theory of computation → Massively parallel algorithms**; • **Computing methodologies → Shared memory algorithms**; **Neural networks**.

*Corresponding author.

KEYWORDS

Winograd, Convolutional Neural Networks, GPU, Performance

ACM Reference Format:
Zhiyi Zhang, Pengfei Zhang, Zhuopin Xu, Bingjie Yan, and Qi Wang. 2024. Im2col-Winograd: An Efficient and Flexible Fused-Winograd Convolution for NHWC Format on GPUs. In *The 53rd International Conference on Parallel Processing (ICPP'24), August 12–15, 2024, Gotland, Sweden.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3673038.3673039

1 INTRODUCTION

Convolutional neural networks (CNNs) play crucial roles in deep learning (DL). For CNNs, convolution computing accounts for the majority of arithmetic complexity, making it a key factor that affects the execution speed. Fast CNN execution is commonly based on parallel processors, such as GPUs, CPUs, and FPGAs, as they can efficiently perform the dot products in convolution.

To address the massive convolution computing in modern CNNs, Winograd [14] algorithm has been used to reduce the number of multiplications. Previous works [2, 3, 14, 17, 19, 26, 34] have implemented fast Winograd 2D convolution on GPUs. They generally use 2D Winograd, and organize tensors in *NCHW* or *CHWN* format rather than *NHWC* to achieve coalesced data loads. Despite this, *NHWC* is one of the most popular tensor formats in CNNs.

Limited by the space complexity of Winograd and the high-speed memory capacity of hardware, previous implementations are commonly designed for specific small filters, potentially lacking flexibility. Typically, fused-Winograd implementations are restricted to 3×3 filters, while the non-fused can support other filter shapes that are usually less than or equal to 5×5.

To tackle these issues, we propose Im2col-Winograd, an efficient and flexible fused-Winograd convolution for *NHWC* format on GPUs. Im2col-Winograd decomposes an ND convolution into a set of 1D convolutions, and then cumulatively performs 1D Winograd on them. Compared to 2D Winograd, this decomposition improves data-access continuity and reduces space complexity, allowing Im2col-Winograd to support more filter shapes (Section 4.2).

We have implemented 3 types of Im2col-Winograd, supporting unit-stride 2D convolution and deconvolution with 2-9 filter widths. All computing stages are fused into one to eliminate the need for auxiliary space. To enhance GPU performance: (1) we redesign the workflow to ensure large cache-block size, consistent performance, and coalesced memory access; (2) we minimize bank conflicts to facilitate loads and stores on shared memory; (3) we simplify data transformations, and optimize kernel functions for particular filter, channel, and padding sizes; (4) we reuse input-tile overlaps to reduce data-loading costs; (5) we coordinate multiple kernels to treat tensor boundaries, thus avoiding unnecessary calculations.

The experiments indicate that Im2col-Winograd achieves 0.788× to 2.05× speedup over the fastest benchmark algorithm in cuDNN [3, 26], while maintaining good accuracy. Additionally, Im2col-Winograd shows comparable convergence to PyTorch [27], when training CNNs on Cifar10 [12] and ILSVRC2012 [5, 16] datasets.

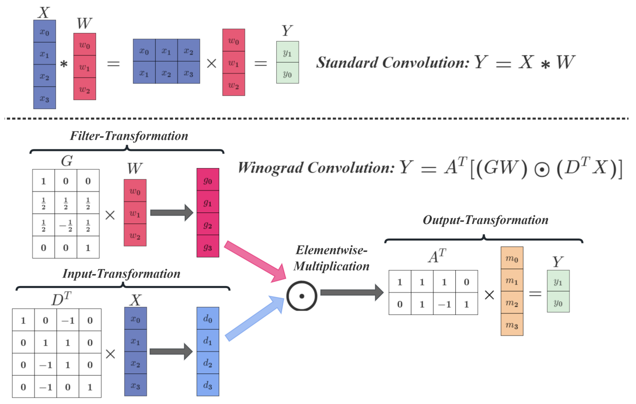

Figure 1: The 1D Winograd and Standard Convolution. This figure presents the workflow of Winograd $F(2,3)$.

2 BACKGROUND

The input and output tensors of convolution are respectively referred to as input and output feature maps (*ifms, ofms*), with filters acting on *ifms* to generate *ofms*. In CNNs, 2D convolutions involve channels and batches, so *ifms*, *ofms*, and filters are 4D tensors.

There are many ways to implement convolution, like direct [35], GEMM [3, 10], FFT [23, 30], and Winograd. GEMM is a variant of direct convolution. FFT is efficient for large filters. Winograd includes fewer multiplications, leading to a lower time expense, since hardware executes multiplication much slower than addition.

The Winograd algorithm was discovered by Toom [29] and Cook [4] and generalized by Winograd [31]. As depicted in Figure 1, this algorithm has 4 stages: *filter-transformation*, *input-transformation*, *elementwise-multiplication* (elem-mul), and *output-transformation*. A, G, and D are transform matrices. W, X, and Y are tiles respectively of filters, *ifms*, and *ofms*. *Fused-Winograd* integrates the last 3 stages into 1 kernel function (kernel), while the *Non-Fused* uses multiple kernels and requires a much larger workspace to store intermediate variables.

To optimize hardware performance, the dot products of convolution are typically transformed into outer products. The outer products enable a group of threads to share the transformed input and filter tiles, reducing the workload for a single thread. When dealing with large channels, the complexity of output-transformation is much lower compared to the other stages. Hence, the time complexity of Winograd primarily arises from the elem-mul stage, and is equivalent to that of elem-mul in ideal conditions.

When using a filter of size r to get n outputs, Winograd $F(n,r)$ performs only $(n+r-1)$ elem-muls, at most reducing the multiplication number to $\frac{n+r-1}{nr}$ of standard convolution. By nesting $F(n,r)$ with itself, the 2D Winograd $F(n \times n, r \times r)$ is obtained. It uses a $r \times r$ filter to generate $n \times n$ outputs, and its formula is $Y = A^T[(GWG^T) \odot (D^TXD)]A$.

Winograd convolution has been implemented on GPUs [2, 3, 14, 17, 19, 26, 34], CPUs [15, 22, 32, 33], and FPGAs [18, 20], mainly designed for unit-stride cases and large data volumes, with 2D Winograd being the mainstream choice for 2D convolution.

3 MOTIVATION

The primary goal of Im2col-Winograd is to reduce the space complexity of Winograd and the access discontinuity in *NHWC* format. The reduced time complexity of Winograd comes with increased space complexity. Because of its larger data-access units but fewer multiplications, Winograd requires loading more items to reach the same outer-product scale as standard convolution. $F(n,r)$ uses $\alpha = (n+r-1)$ variables to record the intermediate states, greater than the n variables of standard convolution. As a result, Winograd demands more memory and registers, and its lower multiplication density poses challenges in hiding memory latency.

On GPUs, fused-Winograd needs a much smaller workspace in global memory than the non-fused, which is beneficial for large models. Fused-Winograd stores transformed tiles in shared memory (SMEM) to execute outer products. Since a streaming multiprocessor (SM) has limited SMEM and registers, the outer-product scale and the state-count[1] α^N of ND Winograd are mutually constrained. Increased α can support larger filters and reduce more multiplications, while greater outer-product scale commonly brings better hardware performance. Therefore, the complexity and applicability of Winograd restrict the hardware efficiency, and vice versa.

When arranging tensors in *NHWC* format, the data access within 2D tiles of 2D Winograd is discontinuous and difficult to be vectorized. This is because the items are spaced at channel sizes, and the distances between different rows are determined by the product of width and channel size. Big channels and feature maps in modern CNNs further exacerbate this challenge. Moreover, due to the limited registers and SMEM, it's challenging to vectorize 2D input tiles along the channel axis in a single thread.

To reduce space complexity, it's essential to minimize α^N and auxiliary space, while ensuring similar time complexity. To reduce data-access discontinuity, high-dimensional tiles should be avoided. Both goals can be achieved, by decomposing an ND convolution into 1D convolutions to use 1D Winograd.

4 DESIGN OVERVIEW

This paper introduces the GPU implementation of Im2col-Winograd for 2D convolutional layers, FP32 datatype, and *NHWC* format. Unless specified, the following introductions are in the view of forward convolution. The math notations are listed in Table 1.

[1]The minimum number of variables to characterize the state of a system.

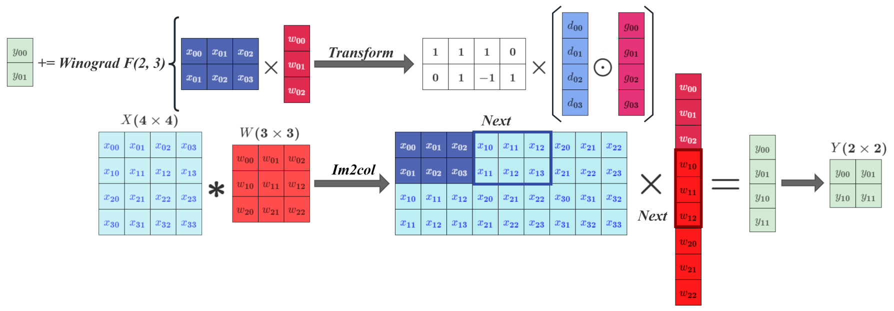

Figure 2: Im2col-Winograd convolution. This figure depicts the workflow of Im2col-Winograd $\Gamma_4(2,3)$.

4.1 Proposed Method

Im2col-Winograd with α state-count and $F(n,r)$ is represented as $\Gamma_\alpha(n,r)$. As shown in Figure 2, $\Gamma_\alpha(n,r)$ consists of 2 stages:

- **Stage1: *Im2col*** The filters W and *ifms* X are respectively transformed into matrices $A \in R^{G_K \times G_N}$, and $B \in R^{G_M \times G_K}$, using Im2col operator, where $G_N = O_C$, $G_M = N * O_H * O_W$, and $G_K = I_C * F_H * F_W$.
- **Stage2: *Winograd*** Use $n \times r$ and $r \times 1$ sliding windows to move across n rows of A and 1 column of B without overlapping. Use $F(n,r)$ to conduct 1D convolution on the items selected by these sliding windows. The convolution results are accumulated to calculate n items of the *ofms* Y.

Since Im2col is an index mapping, these 2 stages are fused into one, thus avoiding auxiliary space to store intermediate variables.

The fast memory on GPUs, like SMEM and registers, is relatively small, so the cache-blocking strategy [13] is adopted to maximize data reuse. To effectively hide memory latency, the cache-block size should be a minimum of $32 \times 32 \times 8$. Specifically, in each iteration, a thread-block (block) at least loads 8 sets of 32 input tiles and 32 filter tiles, and requires $4\alpha(32+32)8$ bytes SMEM to store the transformed tiles (also need SMEM for other purposes). Since the max SMEM for a block is 49152 bytes, α must ≤ 24 and is preferably a power of 2. Consequently, suitable options for α are 4, 8, and 16.

As shown in Figure 3, we implemented 4/8/16 state Im2col-Winograd, for unit-stride 2D convolution and deconvolution with 2-9 filter widths. They are developed in C++ without PTX or SASS (assembly languages of CUDA). Although this approach may not achieve the max hardware efficiency, it offers better portability, readability, and maintainability.

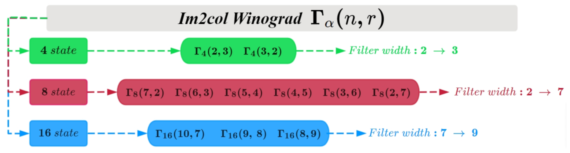
Im2col Winograd $\Gamma_\alpha(n,r)$
4 state
$\Gamma_4(2,3)$ $\Gamma_4(3,2)$
Filter width : 2 → 3
8 state
$\Gamma_8(7,2)$ $\Gamma_8(6,3)$ $\Gamma_8(5,4)$ $\Gamma_8(4,5)$ $\Gamma_8(3,6)$ $\Gamma_8(2,7)$
Filter width : 2 → 7
16 state
$\Gamma_{16}(10,7)$ $\Gamma_{16}(9,8)$ $\Gamma_{16}(8,9)$
Filter width : 7 → 9

Figure 3: 4/8/16 state Im2col-Winograd implementations.

4.2 Advantages over 2D Winograd

Im2col-Winograd is more flexible. It only restricts 1 filter dimension (F_W), unlike 2D Winograd restricts two (F_H and F_W). Given the constraint that $\alpha^N \leq 16$, 2D Winograd supports 2×2 to 3×3 filters, with $F(3\times3, 2\times2)$ to $F(2\times2, 3\times3)$ as the selections; on the other hand, Im2col-Winograd can deal with 2-15 filter widths, offering much more options including $\Gamma_4(3,2)$ to $\Gamma_4(2,3)$, $\Gamma_8(7,2)$ to $\Gamma_8(2,7)$, and $\Gamma_{16}(15,2)$ to $\Gamma_{16}(2,15)$. Furthermore, Im2col-Winograd can be applied to ND convolution, by expanding Stage1 Im2col to ND, while remaining Stage2 unchanged.

Im2col-Winograd can be more lightweight. For instance, the $F(2\times2, 3\times3)$ is adopted in most FP32 fused-Winograd implementation. Both $F(2\times2, 3\times3)$ and $\Gamma_8(6,3)$ reduce the multiplication number to $\frac{1}{2.25}$. However, $F(2\times2, 3\times3)$ uses 4^2 states and loads $\frac{25}{4}$ items per output, while $\Gamma_8(6,3)$ only uses 8 states and loads $\frac{33}{6}$ items per output. Potentially, $\Gamma_8(6,3)$ requires fewer registers and SMEM, and has advantages in hiding memory latency.

The data access is more continuous in 1D tiles of Im2col-Winograd. This also leads to shorter distances between work areas of blocks, enabling data to stay in L2 cache for a longer period.

Table 1: Math Notations of 2D Convolution

Symbol	Meaning
I_H, I_W, I_C	Input height, width, and channel size
O_H, O_W, O_C	Output height, width, and channel size
F_H, F_W	Filter height and width
N	Batch size
X	The *ifms* $X \in R^{N \times I_H \times I_W \times I_C}$
W	The filters $W \in R^{O_C \times F_H \times F_W \times I_C}$
Y	The *ofms* $Y \in R^{N \times O_H \times O_W \times O_C}$
p_H, p_W	Padding size on height and width axis

5 IMPLEMENTATION DETAIL

In our implementations, the zero-padding on *ifms* is implicitly achieved using conditional statements, and the texture memory is utilized to prevent warp divergence and expedite input-tile loading. The elem-muls are transformed into outer products in $8 \times (8 \times 8)$ units, where multiplications and additions are fused into multiply-add instructions to reduce calculating expenses. To mitigate the access discontinuity of *NHWC* format, a warp, consisting of 32 threads, is designed to access items in adjacent channels.

Algorithm 1 The simplified block workflow of $\Gamma_8(n,r)$.
(1) α, B_K, B_N, and B_M respectively equal 8, 8, 64, and 32.
(2) $A_{(\alpha,n)}$, $G_{(\alpha,r)}$, and $D_{(\alpha)}$ are transform matrices of $F(n,r)$.
(3) t_x and t_y are thread indices.

```
Function loadTiles($X, W, Gs, Ds, G_i, G_k, D_i, D_k, buf$)
    $W_{tile}[2][r],\ X_{tile}[\alpha] \leftarrow$ 2 tiles of $W$, 1 tile of $X$
    $G_{tile}[2][\alpha],\ D_{tile}[\alpha] \leftarrow G_{(\alpha,r)}W_{tile},\ D_{(\alpha)}{}^T X_{tile}$
    $Gs[buf][G_k][][G_i : G_i+2] \leftarrow G_{tile}[0:2][]$
    $Ds[buf][D_k][][D_i] \leftarrow D_{tile}[]$
    __syncthreads()
Function outerProduct($v[64], Gs, Ds, G_{Idx}, D_{Idx}, u_x, buf$)
    for ($i_k \leftarrow 8; i_k < 8; i_k \leftarrow i_k + 1$)
        $a[8] \leftarrow Gs[buf][i_k][u_x][G_{Idx} : G_{Idx}+8]$
        $b[8] \leftarrow Ds[buf][i_k][u_x][D_{Idx} : D_{Idx}+8]$
        $v[] \leftarrow v[] + (a[] \times b[])$
    $buf \leftarrow buf$ ^ 1  # switch double buffer, Bitwise XOR
Function transformOutput($v[64], Y, u_x, u_y, Ys$)
    for ($i \leftarrow 0; i < 2; i \leftarrow i + 1$)
        __syncthreads()
        for ($j \leftarrow 0; j < 4; j \leftarrow j + 2$)
            $Ys[u_x][u_y][] \leftarrow \frac{1}{4}$ of $v$  # 16 items
            __syncthreads()
            $Y_{tile}[j+0][0:n] \leftarrow A^T_{(\alpha,n)} Ys[][u_y][2u_x+0]$
            $Y_{tile}[j+1][0:n] \leftarrow A^T_{(\alpha,n)} Ys[][u_y][2u_x+1]$
        store $Y_{tile}[][]$ to $Y$  # 4 tiles

1: __shared__ $Gs[2][B_K][\alpha][B_N],\ Ds[2][B_K][\alpha][B_M]$
2: calculate $G_k,\ G_i,\ D_k,\ D_i,\ u_y,\ u_x$ based on $t_x,\ t_y,\ \alpha$
3: calculate $G_{Idx},\ D_{Idx}$ based on $u_y$
4: $accumulator[64],\ buf \leftarrow \vec{0},\ 0$
5: for ($fh \leftarrow 0; fh < F_H; fh \leftarrow fh + 1$)
6:     loadTiles($X, W, Gs, Ds, G_i, G_k, D_i, D_k, buf$)
7:     for ($oic \leftarrow B_K; oic < I_C; oic \leftarrow oic + B_K$)
8:         outerProduct($accumulator, Gs, Ds, G_{Idx}, D_{Idx}, u_x, buf$)
9:         loadTiles($X, W, Gs, Ds, G_i, G_k, D_i, D_k, buf$)
10:    outerProduct($accumulator, Gs, Ds, G_{Idx}, D_{Idx}, u_x, buf$)
11: __shared__ $Ys[\alpha][\frac{B_N}{2}][16]$  # reuse $Gs$
12: transformOutput($accumulator, Y, u_x, u_y, Ys$)
```

Algorithm 2 The simplified block workflow of $\Gamma_{16}(n,r)$.
(1) α, B_K, B_N, and B_M respectively equal 16, 8, 32, and 32.
(2) and (3) are the same as those of Algorithm 1.

```
Function storeTiles($X, W, Gs, Ds, G_i, G_k, D_i, D_k$)
    $G_{tile}[\alpha],\ D_{tile}[\alpha] \leftarrow G_{(\alpha,r)}W_{tile},\ D^T_{(\alpha)} X_{tile}$
    $Gs[G_k][][G_i],\ Ds[D_k][][D_i] \leftarrow G_{tile}[],\ D_{tile}[]$
    __syncthreads()
Function outerProduct($v[64], Gs, Ds, G_{Idx}, D_{Idx}, u_x$)
    for ($i_k \leftarrow 8; i_k < 8; i_k \leftarrow i_k + 1$)
        $a[8] \leftarrow Gs[i_k][u_x][G_{Idx} : G_{Idx}+8]$
        $b[8] \leftarrow Ds[i_k][u_x][D_{Idx} : D_{Idx}+8]$
        $v[] \leftarrow v[] + (a[] \times b[])$
Function transformOutput($v[64], Y, u_x, u_y, Ys$)
    for ($i \leftarrow 0; i < 4; i \leftarrow i + 2$)
        $Ys[0][u_x][u_y][] \leftarrow \frac{1}{4}$ of $v$  # 16 items
        $Ys[1][u_x][u_y][] \leftarrow \frac{1}{4}$ of $v$  # 16 items
        __syncthreads()
        $Y_{tile}[i+0][0:n] \leftarrow A^T_{(\alpha,n)} Ys[0][][u_y][u_x]$
        $Y_{tile}[i+1][0:n] \leftarrow A^T_{(\alpha,n)} Ys[1][][u_y][u_x]$
    store $Y_{tile}[][]$ to $Y$  # 4 tiles

1: __shared__ $Gs[B_K][\alpha][B_N],\ Ds[B_K][\alpha][B_M]$
2: calculate $G_k,\ G_i,\ D_k,\ D_i,\ u_y,\ u_x$ based on $t_x,\ t_y,\ \alpha$
3: calculate $G_{Idx}, D_{Idx}$ based on $u_y$
4: $accumulator[64] \leftarrow \vec{0}$
5: for ($fh \leftarrow 0; fh < F_H; fh \leftarrow fh + 1$)
6:     $W_{tile}[r],\ X_{tile}[\alpha] \leftarrow$ 1 tiles of $W$, 1 tile of $X$
7:     for ($oic \leftarrow B_K; oic < I_C; oic \leftarrow oic + B_K$)
8:         storeTiles($X, W, Gs, Ds, G_i, G_k, D_i, D_k$)
9:         outerProduct($accumulator, Gs, Ds, G_{Idx}, D_{Idx}, u_x$)
10:        $W_{tile}[r],\ X_{tile}[\alpha] \leftarrow$ 1 tiles of $W$, 1 tile of $X$
11:        __syncthreads()
12:    storeTiles($X, W, Gs, Ds, G_i, G_k, D_i, D_k$)
13:    outerProduct($accumulator, Gs, Ds, G_{Idx}, D_{Idx}, u_x$)
14:    __syncthreads()
15: __shared__ $Ys[2][\alpha][\frac{B_N}{2}][16]$  # reuse $Gs$, $Ds$
16: transformOutput($accumulator, Y, u_x, u_y, Ys$)
```

5.1 Basic Workflow

The tasks of $\Gamma_\alpha(n,r)$ are distributed among $\frac{O_C}{B_N} \times (\frac{NO_H}{B_M}\frac{O_W}{n})$ blocks. This blocking approach is beneficial for consistent performance, since the block number is determined by the dimensions of both feature maps and channels. For example, the early layers of CNNs commonly have small channels but large feature maps, while the end layers have large channels but small feature maps. However, the product of channel size and feature-map size tends to be fair across all layers, so the block number can be consistent.

Algorithm 1 and 2 show the block workflows of $\Gamma_8(n,r)$ and $\Gamma_{16}(n,r)$. The workflow of $\Gamma_4(n,r)$ is analogized to that of $\Gamma_8(n,r)$.

Each block has 16×16 threads, and performs $F_H \times \frac{I_C}{B_K}$ iterations to compute $B_N \times B_M$ output tiles. B_K is 8 for all values of α. $B_N \times B_K$ is 64×64 when α is 4, 64×32 when α is 8, and 32×32 when α is 16. Such $B_N \times B_M \times B_K$ cache-block size is large enough to ensure computing intensity and hide memory latency.

Each thread has 64 accumulators to compute $\frac{1}{\alpha}$ of Winograd states. Via SMEM, every $\frac{256}{\alpha}$ threads with the same u_y share their transformed tiles to execute outer products and update accumulators; every α threads with the same u_x exchange their accumulators to accomplish the output-transformation.

In every iteration, each thread loads $\frac{B_N}{32}$ and $\frac{B_M}{32}$ tiles from filters and *ifms*, transforms and stores these tiles in SMEM, and then executes (8×8) outer products for B_K times. Following the outer-product calculation, threads move on to pre-fetch and transform tiles for the next iteration. The outer products, tile pre-fetching, and tile transformation in different warps are concurrently executed on a SM, enabling the computation and memory access to overlap and hide each other's latency. To achieve $B_N \times B_M \times B_K$ cache-block size, a block of $\Gamma_\alpha(n,r)$ requires $4\alpha(B_N + B_M)B_K$ bytes SMEM. When α

is 4 or 8, the required SMEM $\leq \frac{1}{2}$ of the max SMEM (24576 bytes), so the double-buffered SMEM is constructed to further enhance the warp-level parallelism.

Because of the limited SMEM in a block, the output-transformation is completed over 4 rounds. In each round, individual threads store $\frac{1}{4}$ of their accumulators to SMEM, load 16 items from the transposed SMEM, and calculate $\frac{16}{\alpha}$ output tiles in continuous channels. In order to enhance bandwidth, the output tiles of $\frac{\alpha}{4}$ rounds are merged and written to global memory in 128-bit units.

To further reduce register consumption and simplify calculations, we provide templates for particular filter heights and channel sizes. In the context of unit-stride convolution, $r \times r$ filters are typically used with $\lfloor \frac{r}{2} \rfloor$ padding to maintain the size of feature maps, like 3×3 filters with a padding of 1. Based on this observation, we optimized certain kernels of $\Gamma_\alpha(n, r)$ for $p_W \leq \lfloor \frac{r}{2} \rfloor$.

For convolution in forward propagation, filters are transposed into $F_H \times F_W \times I_C \times O_C$ format, to achieve more vectorized and continuous data loads. The transposition cost is relatively small, since feature maps are typically much larger than filters, especially when dealing with big data volumes. For backward deconvolution, the 180-degree filter-rotation is integrated into filter-transformation. The backward kernels have similar performance to the forward kernels without rearranging filters.

5.2 Reduce Bank Conflicts

The bank conflicts on SMEM negatively effect performance, and can stem from both store and load operations.

In Algorithm 1 and 2, the thread indexed by (t_y, t_x) uses (G_k, G_i), (X_k, X_i) and (u_x, u_y) to store data in SMEM, where:

$$[G_k,\ G_i] = [t_y\%8,\ (2t_x + 1_{t_y>7}) * (B_N/32)]$$
$$[X_k,\ X_i] = [t_x\%8,\ (2t_y + 1_{t_x>7}) * (B_M/32)]$$
$$[u_x,\ u_y] = [\ t_y/\theta,\ 16(t_y\%\theta) + t_x] \text{ with } \theta = 16/\alpha$$

When storing data in Ds and Ys, threads within a warp compete for access to the same bank on SMEM, leading to conflicts. We pad SMEM arrays to distribute store operations across different banks, thus alleviating conflicts. Since the data is stored in 128-bit units, the last dimensions of SMEM arrays are padded to multiples of 4. The padding details are as follows:

$\Gamma_8(n, r)$: $Ys[8][32+1][16+4]$

$\Gamma_{16}(n, r)$: $Ys[2][16][16+1][16+4]$, $Ds[8][16][32+4]$

This padding approach can not be applied to Ds for $\Gamma_8(n, r)$, since Ds and Gs have allocated the maximum SMEM. As an alternative method, we adjust $X_i \leftarrow (X_i + 4X_k)\%32$. Accordingly, in the *outerProduct* function, the $b \leftarrow Ds$ mapping is adjusted to $b[idx] \leftarrow Ds[buf][i_k][u_x][(D_{Idx} + 4i_k + idx)\%32]$.

In *outerProduct* functions, the 128-bit (4 continuous FP32 items) loads from SMEM can lead to bank conflicts. Within a warp, the 32 threads share the same warpIdx, and have distinct laneIdxes from 0 to 31. Based on the laneIdxes, the loading offset (G_{Idx}, D_{Idx}) is modified to prevent these conflicts. The laneIdx arrangement of $\Gamma_8(n, r)$ is illustrated in Figure 4. For a specific thread, (G_{Idx}, D_{Idx}) is calculated as follows:

$$G_{Idx} = 8 * ((u_y\%2) + (u_y/\theta) * 2) \text{ with } \theta = B_M/8$$
$$D_{Idx} = 8 * ((u_y\%\theta)/2)$$

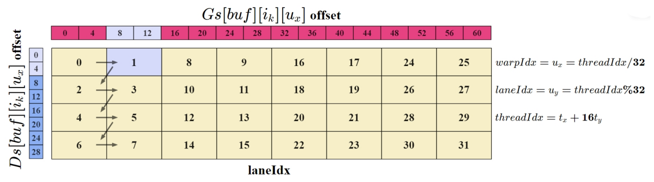

Figure 4: The laneIdx arrangement of $\Gamma_8(n, r)$. To avoid SMEM bank conflicts, *laneIdxes* are arranged in a Z-shape. As illustrated, the *lane*1 thread loads 8-15 items of $Gs[buf][i_k][u_x]$ and 0-7 items of $Ds[buf][i_k][u_x]$, using $2 * 2$ 128-bit loading.

5.3 Simplify Data Transformations

In $\Gamma_8(n, r)$ and $\Gamma_{16}(n, r)$, the data transformations are relatively expensive but can be simplified. The transform matrices of $F(n, r)$ have many solutions. The predominant solution, detailed in Figure 5, is computed using interpolation points at $\{0, 1, -1, 2, -2, \frac{1}{2}, -\frac{1}{2}, 3, -3...\}$. For $A \in R^{\alpha \times n}$, $G \in R^{\alpha \times r}$, and $D^T \in R^{\alpha \times \alpha}$ in this solution, their $(2k+1)^{th}$ and $(2k+2)^{th}$ row vectors have equal items at even positions and opposite items at odd positions, with k starting at 0. Based on this rule, the $(2k+1)^{th}$ and $(2k+2)^{th}$ items of transformed tiles can be calculated together, in which the multiplication outcomes can be reused, thus reducing the number of necessary multiplications by nearly half. The simplified transformations not only reduce the time delay between 2 iterations of outer products, but also improve the computing intensity.

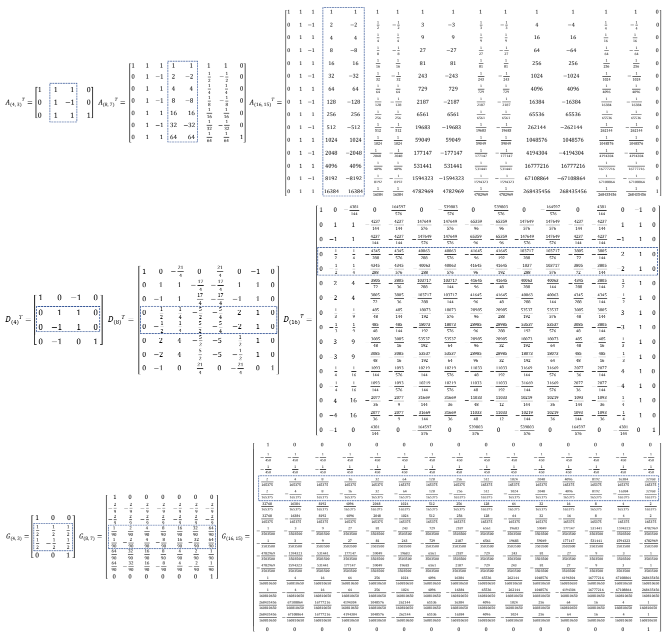

Figure 5: Examples of Winograd Transform Matrices.

5.4 Reuse Input Tile Overlaps

As shown in Figure 6, two adjacent input tiles of $F(n, r)$ have $(r-1)$ overlapping items. This $\frac{r-1}{\alpha}$ overlap can be reused in fast memory

(SMEM, registers) to reduce the access to slower memory (global memory), thus improving speed.

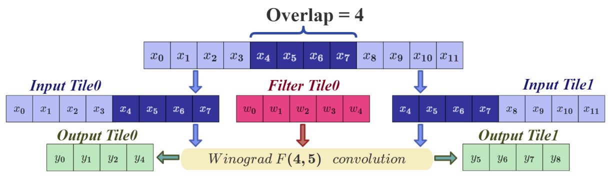
Overlap = 4
x_0 x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8 x_9 x_{10} x_{11}
Input Tile0
Filter Tile0
Input Tile1
w_0 w_1 w_2 w_3 w_4
Output Tile0
y_0 y_1 y_2 y_4
Winograd $F(4,5)$ convolution
Output Tile1
y_5 y_6 y_7 y_8

Figure 6: The Input Tile Overlap. For $F(4,5)$, adjacent input tiles have $(5-1=4)$ overlapping items, as illustrated.

In $\Gamma_4(n,4)$, this overlap can be directly reused, since each thread loads 2 input tiles per iteration. To reuse this overlap in $\Gamma_8(n,r)$ and $\Gamma_{16}(n,r)$ where each thread only loads 1 input tile, the most effective approach we have discovered is merging the tasks of 2 threads into 1 thread. This approach brings both benefits and drawbacks:

- $\uparrow$ **Lower cost of data loading** The average cost of loading 1 input tile is decreased from α items to $(\alpha - \frac{r-1}{2})$ items. The filter data loading is better vectorized in $(\frac{B_M}{16} * 32)$ bit units.
- $\uparrow$ **Higher data-ruse** The outer-product scale increases from $8 \times (8 \times 8)$ to $8 \times (16 \times 8)$, raising the computing intensity and the utilization of high-speed memory.
- $\downarrow$ **Lower parallelism** The thread number per block reduces from 16×16 to 16×8, with each thread using twice as many registers as before. Consequently, the number of active threads decreases, negatively impacting performance.

Based on the experimental findings, the benefits surpass the drawbacks when $\frac{r-1}{\alpha} \geq 0.4375$. Concretely, the data-reuse variants $\Gamma_8^{ruse}(4,5)$, $\Gamma_8^{ruse}(3,6)$, $\Gamma_8^{ruse}(2,7)$, $\Gamma_{16}^{ruse}(9,8)$, and $\Gamma_{16}^{ruse}(8,9)$ have higher performance compared to their initial versions.

5.5 Boundary Treatment

Each output tile of $\Gamma_\alpha(n,r)$ contains n items along the width (O_W) axis. When $O_W\%n \neq 0$ and there is no boundary treatment, these tiles can not exactly cover the *ofms*. If the boundary treatment is implemented using conditional statements, it requires additional registers to check coordinates and causes redundant computations, especially when *ofms* are small. For example, when $\Gamma_8(6,3)$ is used and O_W is 7, there are 2 output tiles along the width axis, and $\frac{5}{6}$ of the computations involved in the second tile is unnecessary.

To address these problems, we achieve the boundary treatment differently, as depicted in Figure 7. The *ofms* are divided into multiple segments along the width axis, with each associated with a distinct kernel. The faster kernel has a higher priority to encompass as large a segment as possible, and this segment can be exactly covered by the kernel. There is no overlap between segments, and the variety of kernels is minimized. GEMM convolution processes the final remaining segment that Im2col-Winograd can not cover.

5.6 Further Enlarge Cache-Block Size

Since many channel sizes in modern CNNs are multiples of 64, and $\Gamma_{16}(n,r)$ still has 16384 bytes SMEM available for allocation, we developed a variant $\Gamma_{16}^{c64}(n,r)$ by enlarging the cache-block size $B_N \times B_M \times B_K$ from $32 \times 32 \times 8$ to $64 \times 32 \times 8$.

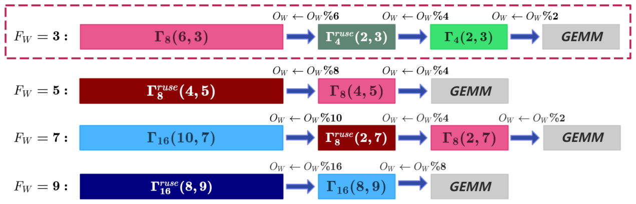

Figure 7: Tensor Boundary Treatment. When ensuring the fastest speed, *ofms* are divided along the width (O_W) axis, and then distributed among multiple kernels. As depicted, when F_W is 3, $\Gamma_8(6,3)$, $\Gamma_4^{ruse}(2,3)$, $\Gamma_4(2,3)$, and *GEMM* are sequentially assigned with the largest divisible part of O_W, $(OW\%6)$, $(OW\%4)$, and $(OW\%2)$ by 6, 4, 2, and 1.

The doubled B_N increases the arithmetic computation intensity (measured in ***operation/byte***) from $\frac{256}{\alpha+r}$ to $\frac{512}{\alpha+2r}$, which is even higher than the $\frac{512}{\alpha+2r+n}$ of $\Gamma_{16}^{ruse}(n,r)$, making $\Gamma_{16}^{c64}(n,r)$ has the best efficiency when dealing with large data volumes. For instance, the intensity of $\Gamma_{16}^{c64}(8,9)$ is 15.06, which is 47.1% higher than the 10.24 of $\Gamma_{16}(8,9)$, and 23.5% higher than the 12.19 of $\Gamma_{16}^{ruse}(8,9)$.

5.7 Integration into Dragon-Alpha

Dragon-Alpha (Alpha) is a Java tensor computing framework that can be used to execute DL algorithms. Cu32, a GPU library of Alpha, is designed for FP32 computations. Alpha and cu32 are fully developed by us, and only require JDK and CUDA for execution.

The kernels of Im2col-Winograd have been merged to form a higher-level encapsulation in cu32, and have been integrated into Alpha-1.2 through the Java native interface. Im2col-Winograd is employed for unit-stride convolution and deconvolution, while other algorithms handle the non-unit-stride cases.

6 EXPERIMENTS

We carried out 3 experiments to evaluate Im2col-Winograd. The 1^{st} compares the performance between Im2col-Winograd and cuDNN. The 2^{nd} accesses the accuracy of Im2col-Winograd, using FP64-CPU convolution as the benchmark. The 3^{rd} uses Im2col-Winograd to train CNNs on Cifar10 and ILSVRC2012, with comparison to PyTorch. Unless specified, the calculations were in FP32 datatype.

In experiment 1 and 2, $\Gamma_\alpha(n,r)$ is performed with $r \times r$ filters and $\lfloor \frac{r}{2} \rfloor$ padding, for $\forall r \in \{2,3,4,5,6,7,8,9\}$. For all test cases, the input-channel size I_C equals the output-channel size O_C.

6.1 Experiment 1: Performance Analysis

To evaluate the performance of Im2col-Winograd, we used it to perform 2D convolution on RTX3060ti and RTX4090 GPUs, which are respectively based on Ampere and Lovelace architectures.

6.1.1 Methods. We use nvcc-11.5 and cuDNN-8.9.0 for RTX3060ti, while nvcc-11.8 and cuDNN-8.9.4 are utilized for RTX4090.

In cuDNN [3], the *Implict_Precomp_GEMM* and *Fused_Winograd* algorithms were used as benchmarks, because they are as memory-efficient as Im2col-Winograd. The *Non_Fused_Winograd* and *FFT* algorithms are unsuitable for benchmarking, since they require a much larger workspace to achieve a much greater reduction

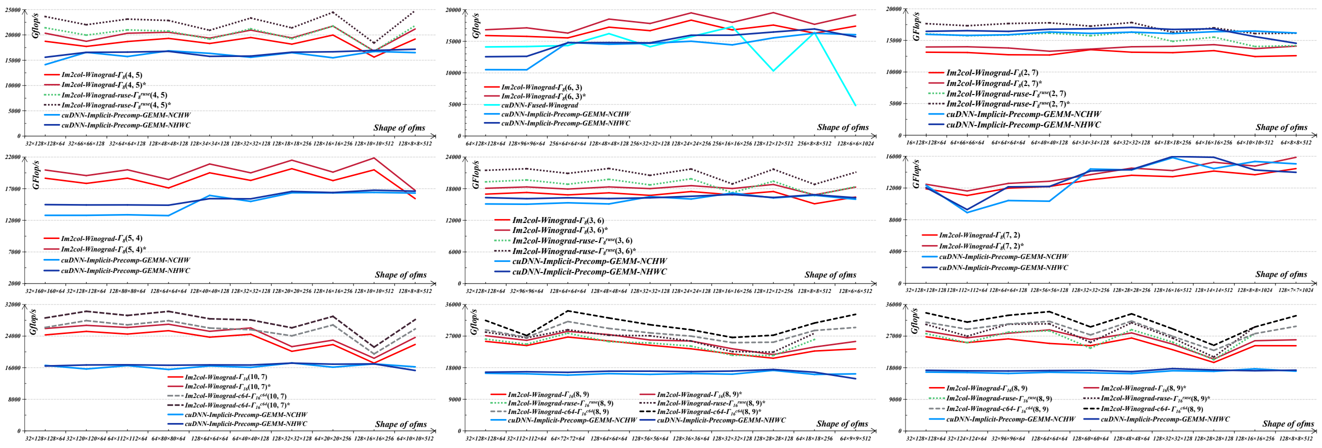
Figure 8: The performance on RTX3060ti. The shapes of *ofms* are in $N \times O_H \times O_W \times O_C$ format.

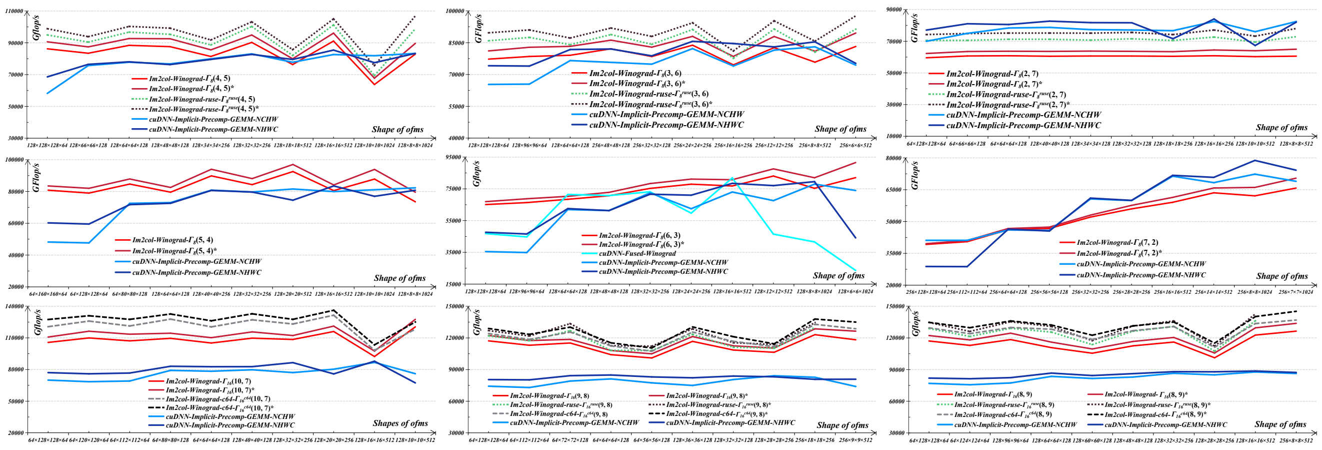
Figure 9: The performance on RTX4090. The shapes of *ofms* are in $N \times O_H \times O_W \times O_C$ format.

in time complexity. The *Fused_Winograd* is restricted to *NCHW* format and 3×3 filters, so it's only comparable to $\Gamma_8(6,3)$. The *Implict_Precomp_GEMM* was executed in both *NHWC* and *NCHW* formats, and it's the fastest algorithm supporting *NHWC* format, which warrants special attention.

We use ***Gflop/s*** to measure performance, denoting 10^9 FP32 operations per second. For a specific convolution, the performance of an algorithm is $\frac{2NO_CO_HO_WF_HF_WI_C}{single\_execution\_time}$. Each algorithm was consecutively executed 1000 times, and the total time was averaged to estimate the *single_execution_time*. Before measurement, each algorithm was executed once to optimize performance.

Table 2: The Speedup over cuDNN. *Fastest* denotes the fastest benchmark algorithm of cuDNN; *NHWC GEMM* denotes the *Implict_Precomp_GEMM* for *NHWC* format.

Algorithm	RTX3060ti		RTX4090	
	Fastest Algorithm	*NHWC GEMM*	*Fastest Algorithm*	*NHWC GEMM*
$\Gamma_8(4,5)$	0.989-1.516×		0.847-1.442×	0.895-1.442×
$\Gamma_8(5,4)$	0.929-1.384×		0.893-1.386×	0.910-1.386×
$\Gamma_8(3,6)$	0.991-1.354×		0.918-1.298×	
$\Gamma_8(6,3)$	0.960-1.221×	0.960-1.358×	0.938-1.477×	0.947-2.074×
$\Gamma_8(2,7)$	0.852-1.076×	0.887-1.110×	0.861-0.968×	0.861-1.087×
$\Gamma_8(7,2)$	0.841-1.243×		0.788-1.034×	0.788-1.428×
$\Gamma_{16}(10,7)$	1.148-1.821×	1.148-1.842×	1.118-1.725×	1.118-1.895×
$\Gamma_{16}(9,8)$	1.445-2.050×	1.445-2.233×	1.293-1.671×	1.293-1.708×
$\Gamma_{16}(8,9)$	1.321-1.976×		1.264-1.664×	

6.1.2 *Analysis.*

Figure 8 and 9 present the performance. '*' means ignoring the time cost of filter-transposition.

According to the results, Im2col-Winograd outperforms cuDNN in most cases, except for $\Gamma_8(7,2)$ and $\Gamma_8(2,7)$. As shown in Table 3, Im2col-Winograd achieves 0.788-2.05× speedup over the fastest benchmark algorithm, and 0.788-2.233× speedup over the *Implict_Precomp_GEMM* for *NHWC* format. Overall, these results highlight the efficiency of Im2col-Winograd.

Compared to cuDNN's *Fused_Winograd*, Im2col-Winograd exhibits more stable performance. Our blocking approach ensures consistent performance, under scenarios of both *large feature maps with small channels* and *small feature maps with large channels*.

Besides, our boundary treatment prevents redundant computations, reducing the impact of feature-map shapes on performance.

The filter transposition is not required in backpropagation. To further improve speed, filters can be pre-transposed before using CNNs for evaluation or prediction. The transposition latency can also be hidden within multi-operator parallelism.

Both $\Gamma_\alpha^{c64}(n,r)$ and $\Gamma_\alpha^{ruse}(n,r)$ show enhanced performance over $\Gamma_\alpha(n,r)$. The enhancement of $\Gamma_\alpha^{c64}(n,r)$ is positive correlated to r, while $\Gamma_\alpha^{ruse}(n,r)$ shows greater enhancement as the $\frac{r-1}{\alpha}$ overlap increases. Due to their better-vectorized data loading and higher computation intensity, they are more robust to L2 cache miss and exhibit more stable performance in cases with large channels.

Due to the boundary treatment method (Section 4.7), $\Gamma_\alpha(n,r)$ has optimal performance when $O_W \% n = 0$; otherwise, the overall performance is compromised by slower algorithms. Generally, the performance exhibits larger fluctuations in intervals with smaller *ofms*, and tends to be smoother as $\frac{n}{\alpha}$ decreases. Since our GEMM convolution used for boundary treatment is slower than cuDNN's GEMM, the performance of Im2col-Winograd may fluctuate relative to cuDNN's. By optimizing the boundary-treatment GEMM, Im2col-Winograd can have a more robust performance.

For $F(n,r)$, its theoretical acceleration Φ is $\frac{nr}{n+r-1}$. Under the constraint that $n+r-1=\alpha$, $\Phi(r) = -\frac{1}{\alpha}r(r-\alpha-1)$ is a convex function and symmetric about $\frac{\alpha+1}{2}$. When α is an odd number, Φ can achieve the global maximum $\frac{1}{\alpha}(\frac{\alpha+1}{2})^2$. However, for even values of α, Φ can only reach the local maximum $\frac{1}{\alpha}\lceil\frac{\alpha+1}{2}\rceil\lfloor\frac{\alpha+1}{2}\rfloor$, when r is $\lceil\frac{\alpha+1}{2}\rceil$ or $\lfloor\frac{\alpha+1}{2}\rfloor$. Additionally, both the lower and upper bounds of Φ increase with α.

The experimental results align with these patterns. $\Gamma_{16}(n,r)$ are generally faster than $\Gamma_8(n,r)$. The performance of $\Gamma_8(n,r)$ is symmetric around $\frac{9}{2}$, and reaches the local maximum when r is 4 or 5. Based on performance, $\Gamma_8(n,r)$ can be divided into 3 levels, where $\Gamma_8(4,5)$ & $\Gamma_8(5,4)$ are the fastest, $\Gamma_8(6,3)$ & $\Gamma_8(3,6)$ have the moderate speed, and $\Gamma_8(7,2)$ & $\Gamma_8(2,7)$ are the slowest. $\Gamma_{16}(n,r)$ achieves the best theoretically acceleration when r is 8 or 9, so $\Gamma_{16}(8,9)$ and $\Gamma_{16}(9,8)$ are faster than $\Gamma_{16}(10,7)$.

With respect to $\frac{\alpha+1}{2}$, the theoretical acceleration of $\Gamma_\alpha(n,r)$ and $\Gamma_\alpha(r,n)$ are symmetric. However, their memory-access overhead is different, resulting in different performance. For instance, $\Gamma_8(6,3)$ has a memory-access overhead that is higher than $\Gamma_8^{ruse}(3,6)$ and lower than $\Gamma_8(3,6)$, so its speed is between the two.

6.2 Experiment 2: Accuracy Analysis

We use FP64-CPU and cuDNN convolutions as benchmarks, to evaluate the accuracy of Im2col-Winograd.

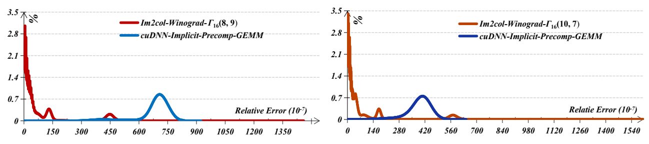

Figure 10: The Distribution of Relative Error.

Table 3: The Average Relative Errors. The shapes of *ofms* are in $N \times O_H \times O_W \times O_C$ format. *CuGEMM* and *CuWinograd* respectively denote the *Implict_Precomp_GEMM* and *Fused_Winograd* algorithms of cuDNN.

Shape of *ofms*	$\Gamma_8(7,2)$	CuGEMM
128 × 112 × 112 × 64	1.43E-07	1.87E-07
128 × 56 × 56 × 128	2.01E-07	2.63E-07
128 × 28 × 28 × 256	2.90E-07	1.30E-05
128 × 14 × 14 × 512	4.31E-07	2.33E-05

Shape of *ofms*	$\Gamma_8(5,4)$	CuGEMM
128 × 80 × 80 × 64	2.09E-07	1.29E-05
128 × 40 × 40 × 128	3.12E-07	2.52E-05
128 × 20 × 20 × 256	4.93E-07	4.67E-05
128 × 10 × 10 × 512	8.28E-07	7.91E-05

Shape of *ofms*	$\Gamma_8(6,3)$	CuGEMM	CuWinograd
128 × 96 × 96 × 64	2.04E-07	1.14E-05	1.09E-07
128 × 48 × 48 × 128	2.69E-07	1.49E-05	1.52E-07
128 × 24 × 24 × 256	3.68E-07	2.92E-05	2.21E-07
128 × 12 × 12 × 512	5.20E-07	5.59E-05	3.37E-07

Shape of *ofms*	$\Gamma_8(2,7)$	CuGEMM
32 × 128 × 128 × 64	2.56E-07	3.93E-05
32 × 64 × 64 × 128	3.80E-07	7.88E-05
32 × 32 × 32 × 256	5.89E-07	7.43E-05
32 × 16 × 16 × 512	9.75E-07	8.92E-05

Shape of *ofms*	$\Gamma_8(4,5)$	CuGEMM
64 × 128 × 128 × 64	2.10E-07	2.02E-05
64 × 64 × 64 × 128	3.05E-07	3.96E-05
64 × 32 × 32 × 256	4.57E-07	7.80E-05
64 × 16 × 16 × 512	7.21E-07	1.45E-04

Shape of *ofms*	$\Gamma_8(3,6)$	CuGEMM
64 × 96 × 96 × 64	2.65E-07	3.08E-05
64 × 48 × 48 × 128	3.99E-07	5.80E-05
64 × 24 × 24 × 256	6.40E-07	1.05E-04
64 × 12 × 12 × 512	1.12E-06	8.62E-05

Shape of *ofms*	$\Gamma_{16}(10,7)$	CuGEMM
64 × 80 × 80 × 64	1.04E-05	3.88E-05
64 × 40 × 40 × 128	1.12E-05	7.60E-05
64 × 20 × 20 × 256	1.27E-05	6.94E-05
64 × 10 × 10 × 512	1.59E-05	1.15E-04

Shape of *ofms*	$\Gamma_{16}(9,8)$	CuGEMM
32 × 144 × 144 × 64	9.86E-06	5.21E-05
32 × 72 × 72 × 128	1.04E-05	1.02E-04
32 × 36 × 36 × 256	1.18E-05	1.89E-04
32 × 18 × 18 × 512	1.48E-05	1.62E-04

Shape of *ofms*	$\Gamma_{16}(8,9)$	CuGEMM
32 × 128 × 128 × 64	9.66E-06	6.83E-05
32 × 64 × 64 × 128	1.02E-05	1.33E-04
32 × 32 × 32 × 256	1.13E-05	2.46E-04
32 × 16 × 16 × 512	1.40E-05	1.35E-04

6.2.1 *Methods.*

The CPU convolution uses FP64 accumulators, providing much higher accuracy than the GPU convolutions. The accuracy is quantified by the average of relative error, and the results of CPU convolution are considered as the true values. The widths of *ofms* are multiples of n to avoid the boundary treatment (Section 4.7). The *ifms* and filters were generated following the uniform distribution within $[1,2]$.

6.2.2 *Analysis.*

The average relative errors are presented in Table3. Figure 10 compares the relative error distribution between $\Gamma_{16}(n,r)$ and cuDNN *Implict_Precomp_GEMM*.

The average relative error of $\Gamma_8(n,r)$ is on the order of 10^{-7}, and that of $\Gamma_{16}(n,r)$ is about 10^{-5}. Compared to cuDNN, Im2col-Winograd has a similar accuracy to *Fused_Winograd*, and a superior accuracy over *Implict_Precomp_GEMM*. As illustrated in Figure 10, compared to *Implict_Precomp_GEMM*, the error distribution of $\Gamma_{16}(n,r)$ is closer to 0 with a smaller average-value. Although $\Gamma_{16}(n,r)$ has a larger maximum-value of error, the proportion of such large values is negligible.

Compared to standard convolution (GEMM, direct), Winograd convolution has fewer multiplications, thus reducing the accuracy loss arising from floating-point operations. With the increase of α, the items in transform matrices of $F(n,r)$ exhibit a larger disparity in their magnitudes. Such disparity can negatively impact accuracy, when it surpasses the precision of a specific datatype. As a result, $\Gamma_{16}(n,r)$ has a lower accuracy compared to $\Gamma_8(n,r)$.

Overall, in most cases, the accuracy of $\Gamma_8(n,r)$ and $\Gamma_{16}(n,r)$ is acceptable for FP32 datatype with a precision of 7 digits.

6.3 Experiment 3: CNN Training

To analyze the efficiency and convergence of Im2col-Winograd, we integrated it into Alpha-v1.2 (mentioned in Section 5.7) to train CNNs [8, 28] on Cifar10 [12] and ILSVRC2012 [5, 16] (a version of ImageNet), using PyTorch-2.2.1 [27] as the benchmark.

6.3.1 *Methods.*

The CNNs of Alpha and PyTorch were identical and underwent the same procedures. Activation functions are LeakyRelu [21]. Specific convolutional and full-connect layers were adjusted to fit tensor shapes, while the backbones of CNNs remain unaltered. 5 BatchNorm [9] layers were added into VGG to expedite convergence. Full-connect and convolutional layers were initialized using kaiming-uniform [7]. SGDM and Adam [11] were used to train CNNs, with SoftMax and 0.001 learning rate.

Cifar10 Input shapes are $32 \times 32 \times 3$ with 10 categories. Batch size is 512. Data was processed on an RTX3060ti.

ILSVRC2012 Input shape are 128×128×3 with 1000 categories. Batch size is 256. Data was processed on an RTX4090.

The labels were encoded to one-hot formats, and the pixels were linearly scaled to $[-1, 1]$. The loss-function value was recorded per 10 steps. To plot the loss curves of ILSVRC2012, a sliding window of size 10 was used to average the loss values without overlap.

VGG16x5 and VGG16x7 are constructed to evaluate $\Gamma_8(4,5)$ and $\Gamma_{16}(10,7)$. Based on the VGG16 [28] architecture, VGG16x5 adjusts all filters from 3×3 to 5×5, and VGG16x7 changes the filter shapes of the first 4 convolutional layers to 7×7.

6.3.2 *Analysis.*

Table 4 and 5 present the performance of Alpha and PyTorch. The loss curves are shown in Figure 11 and 12.

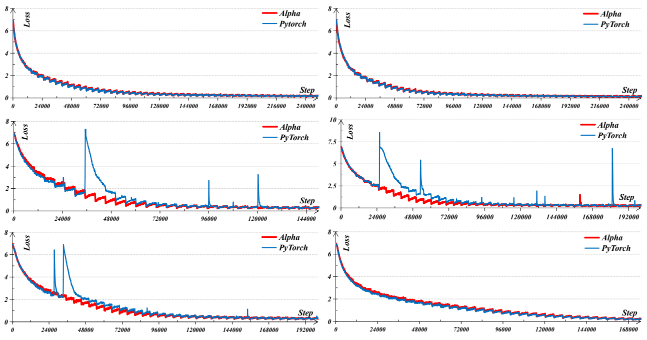

Figure 11: Loss Curves on ILSVRC2012. The training configurations associated with each sub-figure are sequentially enumerated as follows:
0. *ResNet*18 + *Adam*, 50 *epoch*. **1.** *ResNet*34 + *Adam*, 50 *epoch*.
2. *VGG*16 + *Adam*, 30 *epoch*. **3.** *VGG*19 + *Adam*, 40 *epoch*.
4. *VGG*16*x*5 + *Adam*, 40 *epoch*. **5.** *VGG*16*x*7 + *SGDM*, 30 *epoch*.

Table 4: The performance on ILSVRC2012. The data of Alpha is in red, and the data of PyTorch is in blue.

Network	*Training*	*Speed*	*Acceleration*	*Train set accuracy*	*GPU memory*	*Weight file*
ResNet18	*Adam*, 50 epochs	**610.67** s/epoch **922.33** s/epoch	**1.510×**	98.16% 98.30%	**5767** MB **8509** MB	66.8 MB 50.9 MB
ResNet34	*Adam*, 50 epochs	**1028.13** s/epoch **1450.88** s/epoch	**1.411×**	98.35% 98.38%	**9151** MB **12858** MB	117 MB 89.8 MB
VGG16	*Adam*, 30 epochs	**845.63** s/epoch **1172.92** s/epoch	**1.387×**	97.48% 97.65%	**10964** MB **18198** MB	293 MB 223 MB
VGG19	*Adam*, 40 epochs	**925.03** s/epoch **1361.81** s/epoch	**1.472×**	97.86% 97.42%	**11215** MB **18638** MB	320 MB 244 MB
VGG16x5	*Adam*, 40 epochs	**1389.54** s/epoch **2808.32** s/epoch	**2.021×**	96.62% 97.13%	**11308** MB **23750** MB	424 MB 323 MB
VGG16x7	*SGDM,* 30 epochs	**1263.36** s/epoch **2066.59** s/epoch	**1.636×**	95.88% 95.61%	**10794** MB **22176** MB	317 MB 228 MB

On both datasets, CNNs of Alpha and PyTorch show similar convergence and accuracy. However, Alpha outperforms PyTorch in terms of speed and memory usage. When training VGG using PyTorch on ILSVRC012, there are fluctuations in loss curves, primarily attributed to the relatively high learning rate of 0.001.

Based on the results, Im2col-Winograd does not visibly affect the convergence and accuracy of CNNs; its higher accuracy compared to standard convolution can be advantageous for stable convergence. Alpha's lower memory usage demonstrates the memory efficiency of Im2col-Winograd, which is also an advantage for fused convolution algorithms.

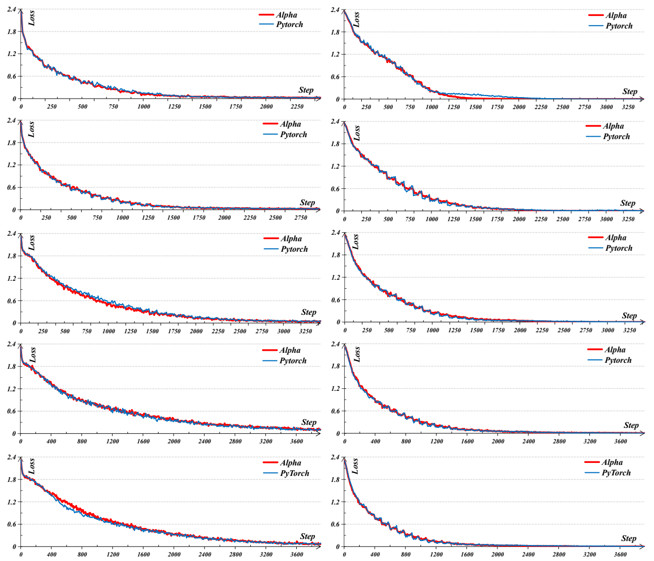

Figure 12: Loss Curves on Cifar10. The training configurations associated with each sub-figure are sequentially enumerated as follows:
0. *ResNet*18 + *Adam*, 25 *epoch*. **1.** *ResNet*18 + *SGDM*, 35 *epoch*.
2. *ResNet*34 + *Adam*, 30 *epoch*. **3.** *ResNet*34 + *SGDM*, 40 *epoch*.
4. *VGG*16 + *Adam*, 35 *epoch*. **5.** *VGG*16 + *SGDM*, 35 *epoch*.
6. *VGG*19 + *Adam*, 40 *epoch*. **7.** *VGG*19 + *SGDM*, 40 *epoch*.
8. *VGG*16*x*5 + *Adam*, 40 *epoch*. **9.** *VGG*16*x*5 + *SGDM*, 40 *epoch*.

Table 5: The performance on Cifar10. The data of Alpha is in red, and the data of PyTorch is in blue.

Network	*Training*	*Speed*	*Acceleration*	*Train\Test accuracy*	*GPU memory*	*Weight file*
ResNet18	*Adam, 25 epochs*	**5.106** s/epoch **5.910** s/epoch	**1.157×**	99.06% \ 78.18% 98.90% \ 77.90%	**1119** MB **2250** MB	66.7 MB 48.2 MB
	SGDM, 35 epochs	**5.076** s/epoch **5.763** s/epoch	**1.135×**	100.0% \ 57.28% 100.0% \ 58.90%	**1054** MB **2246** MB	
ResNet34	*Adam, 30 epochs*	**9.939** s/epoch **11.386** s/epoch	**1.146×**	99.12% \ 79.51% 98.87% \ 79.16%	**1730** MB **3066** MB	120 MB 87.3 MB
	SGDM, 35 epochs	**9.986** s/epoch **11.229** s/epoch	**1.124×**	100.0% \ 60.80% 99.83% \ 61.24%	**1588** MB **2884** MB	
VGG16	*Adam, 35 epochs*	**7.971** s/epoch **9.606** s/epoch	**1.205×**	98.15% \ 83.02% 97.59% \ 82.62%	**1670** MB **3187** MB	78.7 MB 56.7 MB
	SGDM, 35 epochs	**7.942** s/epoch **9.440** s/epoch	**1.189×**	100.0% \ 75.92% 100.0% \ 75.96%	**1602** MB **3172** MB	
VGG19	*Adam, 40 epochs*	**9.784** s/epoch **11.431** s/epoch	**1.168×**	96.08% \ 81.19% 96.03% \ 80.98%	**1812** MB **3252** MB	106 MB 77 MB
	SGDM, 40 epochs	**9.697** s/epoch **11.312** s/epoch	**1.167×**	99.83% \ 76.60% 99.60% \ 76.56%	**1720** MB **3251** MB	
VGG16x5	*Adam, 40 epochs*	**13.944** s/epoch **20.308** s/epoch	**1.454×**	97.88% \ 82.02% 97.65% \ 81.77%	**2046** MB **4976** MB	212 MB 156 MB
	SGDM, 40 epoch	**13.851** s/epoch **19.959** s/epoch	**1.441×**	99.95% \ 77.06% 99.91% \ 77.14%	**1878** MB **4974** MB	

In contrast to VGG, ResNet uses non-unit-stride convolution rather than max-pooling for down-sampling, which restricts the contributions of Im2col-Winograd. That is one of the reasons why Alpha achieves lower acceleration on ResNet compared to VGG. Since $\Gamma_8(4,5)$ and $\Gamma_{16}(10,7)$ result in higher reduction in multiplications over $\Gamma_8(6,3)$, Alpha shows higher acceleration on VGG16x5 and VGG16x7 than VGG16 and VGG19.

In addition to forward convolution and backward deconvolution, the training speed is also related to computing filter gradients, activation functions, data preprocessing, computation scheduling, etc. As a result, the acceleration of Im2col-Winograd may differ from that in Experiment 1. However, given the pivotal role of Im2col-Winograd in CNN training, these results support its efficiency.

7 CONCLUSION

This paper introduces Im2col-Winograd and its GPU implementations for *NHWC* format. The reduced space complexity makes Im2col-Winograd less restricted by the limited resources on a SM, to have better flexibility and applicability. We have implemented the 4/8/16 state Im2col-Winograd for unit-stride convolution and deconvolution with 2-9 filter widths. The effectiveness of our implementations has been demonstrated in experiments.

Compared to previous fused-Winograd implementations, Im2col-Winograd provides acceleration across a broader spectrum of filter shapes, which can be beneficial for investigating CNN structures and extracting features at various scales. In addition to *NHWC* format, our implementations can be ported to *NCHW* and *CHWN* formats while remaining efficiency. Apart from GPU and FP32, the decomposition method and some techniques of Im2col-Winograd may be applicable to other hardware and data types.

The C++ source of Im2col-Winograd is available in the repository of Alpha: https://github.com/GilgameshXYZ123/Dragon-Alpha-v1.2. The code can be modified to cater to more platforms and scenarios.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their valuable comments and suggestions. This work was supported by National Natural Science Foundation of China (No.32070399); HFIPS Director's Fund (No.YZJJKX202201); Anhui Provincial Key Research and Development Program (No.2023n06020016, No.2023n06020028).

REFERENCES

[1] Renzo Andri, Antonio Cipolletta Beatrice Bussolino, Lukas Cavigelli, and Zhe Wang. 2022. Going Further With Winograd Convolutions: Tap-Wise Quantization for Efficient Inference on 4x4 Tiles. In *Proceedings of 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE/ACM, Chicago, IL, USA. https://doi.org/10.1109/MICRO56248.2022.00048

[2] Roberto L. Castro, Diego Andrade, and Basilio B. Fraguela. 2021. OpenCNN: A Winograd Minimal Filtering Algorithm Implementation in CUDA. *Mathematics* 9, 17 (2021). https://doi.org/10.3390/math9172033

[3] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cuDNN: Efficient Primitives for Deep Learning. *CoRR* abs/1410.0759, 2014 (2014).

[4] SA Cook. 1966. *On the minimum computation time for multiplication.* Harvard University.

[5] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. ImageNet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition.* 248–255. https://doi.org/10.1109/CVPR.2009.5206848

[6] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning.* MIT Press, USA. http://www.deeplearningbook.org

[7] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet. In *Proceeding of 2015 IEEE International Conference on Computer Vision (ICCV).* IEEE, 1026–1034. https://doi.org/10.1109/ICCV.2015.123

[8] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR).* IEEE, 770–778. https://doi.org/10.1109/CVPR.2016.90

[9] Sergey Ioffe and Christian Szegedy. 2015. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In *Proceedings of International Conference on Machine Learning (ICML).* ACM, 448–456.

[10] Y. Jia, E. Shelhamer, J. Donahue, and et al. 2014. caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia Retrieval (ICMR).* Newark, NJ, USA, 675–678.

[11] Diederik P. Kingma and Jimmy Lei Ba. 2015. Adam: a Method for Stochastic Optimization. In *Proceedings of the 3rd International Conference on Learning Representations (ICLR).* San Diego, USA.

[12] Alex Krizhevsky. [n. d.]. Cifar10. http://www.cs.toronto.edu/~kriz/cifar.html

[13] Monica D. Lam, Edward E. Rothberg, , and Michael E. Wolf. 1991. The Cache Performance and Optimizations of Blocked Algorithms.. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS).* ACM, Santa Clara, CA, USA.

[14] Andrew Lavin and Scott Gray. 2016. Fast algorithms for convolutional neural networks. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR).* IEEE, Las Vegas, NV, USA. https://doi.org/10.1109/CVPR.2016.435

[15] Dosheng Li, Dan Huang, Zhiguang Chen, and Yutong Lu. 2021. Optimizing Massively Parallel Winograd Convolution on ARM Processor. In *Proceedings of the 50th International Conference on Parallel Processing.* ACM, 1–12. https://doi.org/10.1145/3472456.3472496

[16] Feifei Li. [n. d.]. IMAGENET. https://image-net.org

[17] Guangli Li, Lei Liu, Xueying Wang, Xiu Ma, and Xiaobing Feng. 2020. Lance: efficient low-precision quantized winograd convolution for neural networks based on graphics processing units. In *Proceedings of 2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP).* IEEE, Barcelona, Spain, 3842–3846. https://doi.org/10.1109/ICASSP40776.2020.9054562

[18] Yun Liang, Liqiang Lu, Qingcheng Xiao, and Shengen Yan. 2020. Evaluating Fast Algorithms for Convolutional Neural Networks on FPGAs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 4 (2020), 857–870. https://doi.org/10.1109/TCAD.2019.2897701

[19] Junhong Liu, Dongxu Yang, and Junjie Lai. 2021. Optimizing Winograd-Based Convolution with Tensor Cores. In *Proceedings of the 50th International Conference on Parallel Processing (ICPP).* ACM, Lemont, IL, USA, 1–10. https://doi.org/10.1145/3472456.3472473

[20] Jinming Lu, Hui Wang, Jun Lin, and Zhongfeng Wang. 2024. WinTA: An Efficient Reconfigurable CNN Training Accelerator With Decomposition Winograd. *IEEE Transactions on Circuits and Systems I: Regular Papers* 71, 2 (2024), 634–645.

[21] A. L. Maas, A. Y, Hannun, and A. Y. Ng. 2013. Rectifier Nonlinearities Improve Neural Network Acoustic Models. In *Proceedings of International Conference on Machine Learning (ICML).*

[22] Partha Maji, Andrew Mundy, Ganesh Dasika, Jesse Beu, Matthew Mattina, and Robert Mullins. 2019. Efficient Winograd or Cook-Toom Convolution Kernel Implementation on Widely Used Mobile CPUs. In *2019 2nd Workshop on Energy Efficient Machine Learning and Cognitive Computing for Embedded Applications (EMC2).* 1–5. https://doi.org/10.1109/EMC249363.2019.00008

[23] Michael Mathieu, Mikael Henaff, and Yann LeCun. 2014. Fast Training of Convolutional Networks through FFTs. (2014). https://arxiv.org/abs/1312.5851

[24] NVIDIA. 2024. *CUDA C Best Practices Guide.* NVIDIA, USA. https://docs.nvidia.com/cuda/cuda-c-best-practices-guide

[25] NVIDIA. 2024. *CUDA C Programming Guide.* NVIDIA, USA. https://docs.nvidia.com/cuda/cuda-c-programming-guide

[26] NVIDIA. 2024. NVIDIA cuDNN. https://developer.nvidia.com/cudnn

[27] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, JunjieBai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Proceedings of the 33rd Annual Conference on Neural Information Processing Systems (NIPS).* MIT Press, Vancouver, Canada, 675–678.

[28] Karen Simonyan and Andrew Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *Proceedings of the 3rd International Conference on Learning Representations (ICLR).*

[29] Andrei L Toom. [n. d.]. The complexity of a scheme of functional elements realizing the multiplication of integers. *Soviet Mathematics Doklady* 3 ([n. d.]), 714–716.

[30] Nicolas Vasilache, Jeff Johnson, Michael Mathieu, Soumith Chintala, Serkan Piantino, and Yann LeCun. 2015. Fast Convolutional Nets with fbfft: A GPU Performance Evaluation. In *Proceedings of the 3rd International Conference on Learning Representations (ICLR).* San Diego, CA.

[31] Shmuel Winograd. 1980. *Arithmetic complexity of computations.* Vol. 33. Siam, Salt Lake City, UT.

[32] Ruofan Wu, Feng Zhang, Jiawei Guan, Zhen Zheng, Xiaoyong Du, and Xipeng Shen. 2021. DREW: Efficient Winograd CNN Inference with Deep Reuse. In *Proceedings of the ACM Web Conference 2022April 2022.* ACM, Lyon, France, 1807–1816. https://doi.org/10.1145/3485447.3511985

[33] Da Yan, Wei Wang, and Xiaowen Chu. 2018. Optimizing N-dimensional, winograd-based convolution for manycore CPUs. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP).* ACM, 109–123. https://doi.org/10.1145/3178487.3178496

[34] Da Yan, Wei Wang, and Xiaowen Chu. 2020. Optimizing Batched Winograd Convolution on GPUs. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP).* ACM, San Diego, CA, USA, 32–44. https://doi.org/10.1145/3332466.3374520

[35] Jiyuan Zhang, Franz Franchetti, and Tze Meng Low. 2018. High Performance Zero-Memory Overhead Direct Convolutions. In *Proceedings of the 35th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 80),* Jennifer Dy and Andreas Krause (Eds.). PMLR, 5776–5785.