WinRS: Accelerate Winograd Backward-Filter Convolution with Tiny Workspace

Zhiyi Zhang*
gilgamesh@mail.ustc.edu.cn

Junshi Chen*
cjuns@ustc.edu.cn

Jingwei Sun*
sunjw@ustc.edu.cn

Pengfei Zhang†
pfzhang@aiofm.ac.cn

Zhuopin Xu†
xuzp@iim.ac.cn

Jun Shi*
shijun18@ustc.edu.cn

Qi Wang†‡
wangqi@ipp.ac.cn

Abstract

Winograd algorithm powerfully accelerates Convolutional Neural Networks. However, for backward-filter convolution (BFC), existing implementations often struggle to achieve both high throughput and low memory usage, due to challenges from large filters and small output sizes. This work proposes WinRS, a fast, memory-efficient, and flexible BFC algorithm. WinRS reduces N-D large filters into 1D formats and precisely splits them to match the fastest kernels. These fully-fused kernels execute the entire BFC in on-chip memory with tiny workspace, and exploit the superior acceleration potential of 1D Winograd convolution. To address low parallelism from small outputs, WinRS adaptively balances workloads into an optimal number of block groups, maximizing hardware utilization. WinRS supports various filter-gradient widths (multiples of 2 to 9). When ported to FP16 on Tensor Cores, WinRS achieves 3.27× the throughput of its FP32 CUDA-Core version. In experiments, WinRS achieves 1.05× to 4.7× speedup over cuDNN GEMM using comparable workspace; WinRS uses less than 4% workspace of cuDNN FFT and Winograd, and exhibits higher throughput with memory- and FLOP-bound workloads.

CCS Concepts

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

Keywords

Winograd, Decomposition, Backward-Filter Convolution, Convolutional Neural Networks, GPU

*University of Science and Technology of China, Hefei, China
†Hefei Institutes of Physical Science, Chinese Academy of Sciences, Hefei, China
‡Corresponding author.

ICPP '25, September 08–11, 2025, San Diego, CA

ACM ISBN 979-8-4007-2074-1
https://doi.org/10.1145/3754598.3754599

1 Introduction

Convolutional Neural Networks (CNNs) have played a pivotal role in deep learning, achieving impressive performance in computer vision, natural language processing, scientific computing, and beyond. As models grow increasingly complex to enhance accuracy, CNN training often demands hundreds of GPU hours and dozens-of-gigabytes memory. Backward-filter convolution (BFC) is a primary step in this process. BFC computes filter gradients to update model weights, and accounts for a substantial portion (e.g. ~ 1/3) of overall time complexity. Forward convolution (FC) and backward-data convolution (BDC) are also essential, with all three steps dominating both training time and memory usage.

GEMM [1, 2], Winograd [3–11], and FFT [5, 12, 13] are efficient convolution algorithms, but each faces inherent trade-offs. GEMM is memory-efficient but can be slow due to cubic time complexity. FFT and Winograd reduce time complexity but require large workspace for intermediate results, several times the data size. To overcome this memory bottleneck, prior studies fused multiple Winograd steps into one, developing fused-Winograd [3–6, 9–11, 14], which is both fast and memory-efficient. While effective for FC and BDC with small filters[1] and large output sizes, fused-Winograd has not been thoroughly optimized for BFC due to two challenges: (1) large filters do not match smaller Winograd transform matrices; (2) small output sizes lead to low parallelism and hardware underutilization. Moreover, (3) many Winograd approaches are limited to specific filter shapes, and (4) their GPU implementations often rely on CUDA Cores [3, 4, 6, 14–16] rather than higher-throughput Tensor Cores. This underscores the need for higher flexibility and performance.

To address limitations of existing BFC methods, this work proposes WinRS, a fast, memory-efficient, and flexible Winograd BFC algorithm. WinRS tackles challenges in BFC, by supporting large filters, maximizing parallelism, and leveraging hardware acceleration. We implement WinRS through two main components:

- **Adaptive Configuration**: Via dimension Reduction and filter Split, WinRS bridges N-D large filters with 1D Winograd transforms, precisely matching the fastest kernels. WinRS adaptively optimizes workload distribution based on BFC parameters and hardware, ensuring high parallelism with minimized partitioning overhead in small-output scenarios.

[1]Filters are also known as convolutional kernels. To avoid confusion with 'kernel (kernel function)', we use 'filter' instead of 'convolutional kernel'.

- **High-Performance Kernels**: WinRS kernels fuse all stages to minimize workspace and data movement. These kernels utilize 1D Winograd convolutions for higher acceleration than 2D approaches [3–11, 15–25], and are highly optimized for GPUs with Tensor-Core acceleration.

To cover standard and extended CNN configurations, WinRS utilizes 13 distinct Winograd convolutions, supporting filter gradients with arbitrary heights and widths ranging from 2× to 9×,

We evaluate WinRS on L40S, RTX 4090, 3090 and A5000 GPUs. Experimental results demonstrate that WinRS achieves 1.05× to 4.7× speedup over cuDNN GEMM, using comparable workspace. WinRS requires less than 4% workspace of cuDNN FFT and Winograd, and has higher throughput in memory-bound and compute-intensive cases. The accuracy of WinRS is comparable to cuDNN.

Our contributions are summarized as follows:

- We propose WinRS algorithm, that flexibly accelerates BFC by reducing time complexity by 1.5× to 4.5×, with a small average workspace 18% of data size.
- We introduce a general reduction-split method and an adaptive workload-distribution method, to optimize Winograd convolution for large filters and small outputs.
- We implement fully-fused decomposition 1D-Winograd convolutions on Tensor Cores, with mixed-precision transforms and scaling matrices to enhance numerical stability.

2 Background

2.1 Winograd Convolution

Winograd convolution [3] employs Winograd-minimal-filtering algorithm [26] to reduce time complexity. 1D Winograd convolution $F(n, r)$, formulated in (1), convolves input tile $\mathcal{X} \in \mathbb{R}^{\alpha}$ with filter tile $\mathcal{W} \in \mathbb{R}^{r}$ to generate output tile $\mathcal{Y} \in \mathbb{R}^{n}$, where $\alpha = n + r - 1$. $\mathbf{A} \in \mathbb{R}^{\alpha \times n}$, $\mathbf{G} \in \mathbb{R}^{\alpha \times r}$, and $\mathbf{D} \in \mathbb{R}^{\alpha \times \alpha}$ are transform matrices.

$$\mathcal{Y} = \mathbf{A}^T[(\mathbf{G}\mathcal{W}) \odot (\mathbf{D}^T\mathcal{X})] \tag{1}$$

Nesting $F(n_0, r_0)$ and $F(n_1, r_1)$ yields 2D Winograd convolution $F(n_0 \times n_1, r_0 \times r_1)$, which convolves $\mathcal{X} \in \mathbb{R}^{\alpha_0 \times \alpha_1}$ with $\mathcal{W} \in \mathbb{R}^{r_0 \times r_1}$ to produce $\mathcal{Y} \in \mathbb{R}^{n_0 \times n_1}$, as formulated in (2).

$$\mathcal{Y} = \mathbf{A}_0^T[(\mathbf{G}_0\mathcal{W}\mathbf{G}_1^T) \odot (\mathbf{D}_0^T\mathcal{X}\mathbf{D}_1)]\mathbf{A}_1 \tag{2}$$

Winograd convolution consists of four steps, as shown in (1) and (2): $\hat{\mathcal{W}} = \mathbf{G}\mathcal{W}$ or $\mathbf{G}_0\mathcal{W}\mathbf{G}_1{}^T$ is **filter transform (FT)**; $\hat{\mathcal{X}} = \mathbf{D}^T\mathcal{X}$ or $\mathbf{D}_0^T\mathcal{X}\mathbf{D}_w$ is **input transform (IT)**; $\hat{\mathcal{Y}} = \hat{\mathcal{W}} \odot \hat{\mathcal{X}}$ is **element-wise multiplication (EWM)**; $\mathcal{Y} = \mathbf{A}^T\hat{\mathcal{Y}}$ or $\mathbf{A}_0^T\hat{\mathcal{Y}}\mathbf{A}_1$ is the **output transform (OT)**. FFT convolution [5, 12, 13] has similar four steps, but utilizes Fast Fourier Transform instead of transform matrices. Fused-Winograd [3–6, 9–11, 14] fuses IT, EWM, and OT into one kernel. Non-Fused-Winograd [5, 7, 10, 18, 19, 23] and FFT execute these four steps in separate kernels.

2.2 Challenges of Backward-Filter Convolution

FC, BDC, and BFC are three primary operations of a convolutional layer, each constituting about one-third of the overall time complexity. In each training step: FC convolves input feature maps $\mathbf{X}$ with filters $\mathbf{W}$ to produce output feature maps $\mathbf{Y}$; BDC convolves output gradients $\nabla\mathbf{Y}$ with $\mathbf{W}^T$ to solve input gradients $\nabla\mathbf{X}$; BFC convolves $\mathbf{X}$ with $\nabla\mathbf{Y}$ to compute filter gradients $\nabla\mathbf{W}$.

Modern CNNs [27–30] typically have small filters from 2×2 to 11×11, but high resolutions like 224×224 for ImageNet [31] and 32×32 for Cifar10 [32]. As a result, FC and BDC have small filters and large outputs, whereas BFC operates on large filters and small outputs, as shown in Figure 1. This unique characteristic of BFC poses two challenges for optimizing Winograd convolution.

Challenge 1. The large filters ($\nabla\mathbf{Y}$) mismatch transform matrices, whose dimensions are commonly smaller than 16 for two reasons: (1) More complex elements in larger transform matrices can harm numerical accuracy. In $\mathbf{D} \in \mathbb{R}^{4\times4}$, non-zero elements are simply ±1; however, $\mathbf{D} \in \mathbb{R}^{16\times16}$ involves $\frac{164597}{576}$ and $\frac{1}{16}$. (2) Kernels require on-chip memory (e.g. shared memory and registers) to rapidly perform Winograd transforms and enlarge cache-block sizes [33] for latency hiding. However, the limited on-chip memory constrains both transform-matrix and cache-block sizes.

Challenge 2. The small outputs ($\nabla\mathbf{W}$) impede full hardware utilization. Typical convolution kernels [1, 3, 4, 6–8, 13–15, 34] distribute workload across multiple blocks, with the block count proportional to $\frac{\text{output size}}{\text{cache-block size}}$. As shown in Figure 2, while this blocking approach generates abundant blocks for FC and BFC, the BFC block count is often far fewer than the number of streaming-multi-processors (e.g. 128 on RTX 4090 GPU), which leaves many processors idle and causes performance bottlenecks.

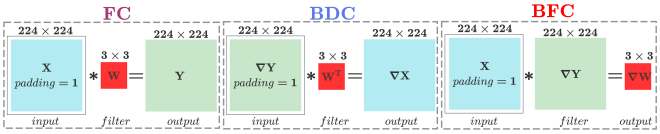

Figure 1: The 2nd convolutional layer in VGG16. The FC and BDC have 3×3 filters and 224×224 outputs; conversely, the BFC has 224×224 filters and 3×3 outputs.

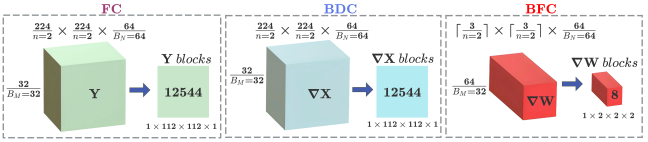

Figure 2: Block count of the 2nd convolution Layer in VGG16. With a cache-block size of $B_N(64) \times B_M(32) \times 8$ and a batch size of 32, the $F(2 \times 2, 3 \times 3)$ kernel [4, 6] yields 12544 blocks for the FC and BDC, but only 8 for the BFC.

3 Algorithm Overview

As a BFC algorithm, WinRS convolves input feature maps $\mathbf{X}$ with output gradients $\nabla\mathbf{Y}$ to compute filter gradients $\nabla\mathbf{W}$. Let $\Omega_\alpha(n, r)$ denote a WinRS kernel using $F(n, r)$ Winograd convolution, where $\alpha = n + r - 1$. The notations of 2D BFC are listed in Table 1.

Before execution, WinRS configures three critical parameters: the fastest pair of kernels, $\Omega_{\alpha_0}(n_0, r_0)$ and $\Omega_{\alpha_1}(n_1, r_1)$; the optimal segment count Z; and the expected segment shape $\widehat{S_H} \times \widehat{S_W}$. As shown in Figure 3, the execution follows a three-phase pipeline:

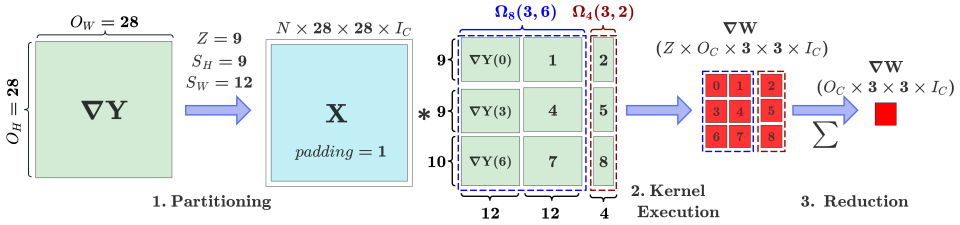

Figure 3: WinRS workflow. 1. Divide ∇Y into 9 segments. The 12 and 4 segment widths are multiples of 6 and 2 to match $F(3,6)$ and $F(3,2)$. 2. The $\{0,1,3,4,6,7\}$-th segments launch $\Omega_8(3,6)$, and the $\{2,5,8\}$-th segments launch $\Omega_4(3,2)$. Figure 4 shows the workflow of $\Omega_8(3,6)$ on the 0-th segment. 3. Results of segments are written to 9 $\nabla\widehat{\mathbf{W}}$ buckets, and summed to compute ∇W.

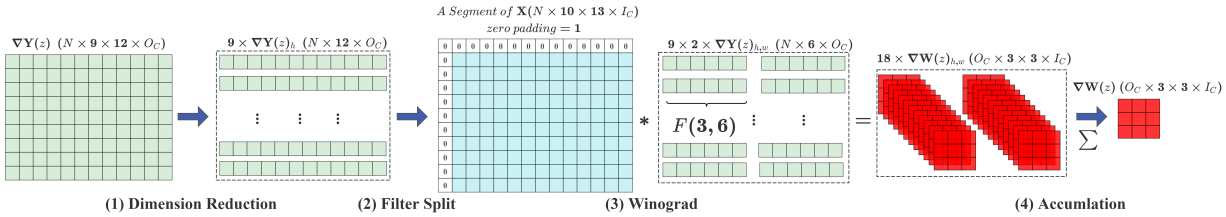

Figure 4: Kernel $\Omega_8(3,6)$ workflow on the z-th segment ($z = 0$ for Figure 3). (1) Decompose $\nabla\mathbf{Y}(z)$ into 9 filters $\in \mathbb{R}^{N\times 12\times O_C}$. (2) Split each filter into 2 units $\in \mathbb{R}^{N\times 6\times O_C}$. (3) Perform $F(3,6)$ Winograd convolution between each unit and its corresponding part of X. (4) Accumulate 18 convolution results to compute the results ∇W(z) of the z-th segment.

Table 1: The Notations of 2D Backward-Filter Convolution.

Symbol	Meaning
I_H, I_W	Input height and width
O_H, O_W	Output-gradient height and width
F_H, F_W	Filter-gradient height and width
I_C, O_C	Input and output channel sizes
N	Batch size
X	Input feature maps $\in \mathbb{R}^{N\times I_H\times I_W\times I_C}$
∇**Y**	Output gradients $\in \mathbb{R}^{N\times O_H\times O_W\times O_C}$
∇**W**	Filter gradients $\in \mathbb{R}^{O_C\times F_H\times F_W\times I_C}$

1. **Partitioning:** Divide $\nabla\mathbf{Y}$ into Z segments, with the z-th segment denoted as $\nabla\mathbf{Y}(z) \in \mathbb{R}^{N\times S_H(z)\times S_W(z)\times O_C}$. To balance workload, the width $S_W(z)$ and height $S_H(z)$ are approximate to $\widehat{S}_H$ and $\widehat{S}_W$. $S_W(z)$ is a multiple of either r_0 or r_1, precisely matching $\Omega_{\alpha_0}(n_0,r_0)$ or $\Omega_{\alpha_1}(n_1,r_1)$.
   Allocate a workspace $(Z-1)$ times the size of ∇**W**. Logically concatenate this workspace with ∇**W** to create $\nabla\widehat{\mathbf{W}}$, which contains Z buckets with the same dimensions as ∇**W**.
2. **Kernel Execution:** Launch $\Omega_{\alpha_0}(n_0,r_0)$ and $\Omega_{\alpha_1}(n_1,r_1)$ on segments with $S_W(z)$ in multiples of r_0 and r_1, respectively. On $\nabla\mathbf{Y}(z)$, the kernel $\Omega_\alpha(n,r) \in \{\Omega_{\alpha_0}(n_0,r_0), \Omega_{\alpha_1}(n_1,r_1)\}$ distributes the workload across a group of blocks, and executes four stages as shown in Figure 4:

(1) **Dimension Reduction:** Decompose $\nabla\mathbf{Y}(z)$ into $S_H(z)$ 1D filters $\in \mathbb{R}^{N\times S_W(z)\times O_C}$, with h-th filter denoted as $\nabla\mathbf{Y}(z)_h$.

(2) **Filter Split:** Split each $\nabla\mathbf{Y}(z)_h \subset \nabla\mathbf{Y}(z)$ into $\frac{S_W(z)}{r}$ units $\in \mathbb{R}^{N\times r\times O_C}$, where the w-th unit is denoted as $\nabla\mathbf{Y}(z)_{h,w}$.

(3) **Winograd:** For each $\nabla\mathbf{Y}(z)_{h,w} \subset \nabla\mathbf{Y}(z)$, perform $F(n,r)$ Winograd convolution between $\nabla\mathbf{Y}(z)_{h,w}$ and the corresponding region of **X**.

(4) **Accumulation:** Accumulate the convolution results from all $\nabla\mathbf{Y}(z)_{h,w} \subset \nabla\mathbf{Y}(z)$ to compute the partition result $\nabla\mathbf{W}(z)$, and write $\nabla\mathbf{W}(z)$ to the z-th $\nabla\widehat{\mathbf{W}}$ bucket.

3. **Reduction:** After all segment results have been written to corresponding buckets, launch a reduction kernel to sum all $\nabla\mathbf{W}$ buckets to calculate $\nabla\mathbf{W}$.

The core objective of WinRS is to achieve both high throughput and low memory usage, while handling large filters and small outputs. To this end, WinRS performs a three-level decomposition:

Level 1. WinRS partitions $\nabla\mathbf{Y}$ into Z segments to distribute the workload across Z block groups, where the block count per group is proportional to $\frac{\text{output size}}{\text{cache-block size}}$. Compared to typical blocking approaches, this partitioning improves parallelism by Z times and enables full hardware utilization.

Level 2. WinRS reduces $\nabla\mathbf{Y}(z)$ into 1D filters, enabling straightforward extension to N-D BFC with two modifications: in Partitioning, divide $\nabla\mathbf{Y} \in \mathbb{R}^{N\times D_1\times\ldots\times D_k\times O_C}$ into Z segments; in Dimension Reduction, decompose $\nabla\mathbf{Y}(z) \in \mathbb{R}^{N\times S_1(z)\times\ldots\times S_k(z)\times O_C}$ into $\frac{\prod_{i=1}^{k} S_i(z)}{S_k}$ filters $\in \mathbb{R}^{N\times S_k(z)\times O_C}$. The 1D filters enable the use of 1D Winograd convolution, which has a higher acceleration upper limit and greater optimization ease than 2D Winograd. Relative to direct convolution, $F(n,r)$ and $F(n_0\times n_1, r_0\times r_1)$ achieve acceleration factors[2] of $A_{1D} = \frac{nr}{\alpha}$ and $A_{2D} = \frac{n_0 r_0}{\alpha_0}\frac{n_1 r_1}{\alpha_1}$. As formulated in (3), with an equivalent space-complexity limit ($\alpha = \alpha_0\alpha_1$), A_{1D} has a higher upper limit. As formulated in (4), for a cache-block size

[2]Direct convolution requires nr multiplications, while $F(n,r)$ performs only α multiplications in EWM——the primary source of time complexity.

of $B_N \times B_M \times B_K$, fused $F(n, r_0)$ kernels have higher computation intensity (ρ_{1D}) than that (ρ_{2D}) of $F(n_0 \times n_1, r_0 \times r_1)$, leading to superior computing efficiency.

$$A_{1D}^{max} = \left(\frac{\alpha+1}{2\alpha}\right)^2 \geq A_{2D}^{max} = \left(\frac{\alpha_0+1}{2\alpha_0}\frac{\alpha_1+1}{2\alpha_1}\right)^2 \quad (\alpha = \alpha_0\alpha_1) \tag{3}$$

$$\rho_{1D} = \frac{2B_N B_M}{B_N r_0 + B_M \alpha} \geq \rho_{2D} = \frac{2B_N B_M}{B_N r_0 r_1 + B_M \alpha_0 \alpha_1} \quad (\alpha = \alpha_0\alpha_1) \tag{4}$$

Level 3. By splitting 1D filters into hybrid units, WinRS indirectly matches large filters with much smaller transform matrices. This process also reduces the space complexity required for convolution, enabling WinRS kernels to fuse all stages into one. The full fusion minimizes workspace and global-memory data movement for intermediate results, allowing all stages to be executed rapidly in on-chip memory. This filter split avoids zero padding [22, 24] to eliminate redundant calculations.

4 Configuration Adaptation

To optimize efficiency, WinRS adjusts execution configurations in three steps: (1) select the fastest pair of WinRS kernels, (2) estimate the baseline segment count; (3) calculate suitable segment shapes and the final segment count.

4.1 Fastest Kernel Pair Selection

To split a 1D filter into hybrid units without zero padding, WinRS requires at least two distinct unit widths, corresponding to two different 1D Winograd convolutions. Consequently, WinRS selects the fastest pair of kernels, $\Omega_{\alpha_0}(n_0, r_0)$ and $\Omega_{\alpha_1}(n_1, r_1)$, to perform BFC, prioritizing the higher-throughput kernel $\Omega_{\alpha_0}(n_0, r_0)$ for the bulk of computations. This kernel pair satisfies three criteria: (1) n_0 and n_1 are divisors of F_W; (2) there exists integers k_0 and k_1, such that $k_0 r_0 + k_1 r_1 = O_W$; (3) it achieves maximal theoretical throughput, determined by weighting kernel throughput coefficients.

Figure 5 shows examples of the fastest kernel pairs. When $F_W = 3$ and $O_W = 16$, WinRS selects $\Omega_8(3,6)$ and $\Omega_4(3,2)$. The higher-throughput kernel $\Omega_8(3,6)$ processes portions of O_W divisible by 6, while $\Omega_4(3,2)$ handles the residual components. Although more than two kernels could improve throughput stability, the marginal gains are outweighed by increased partitioning overhead.

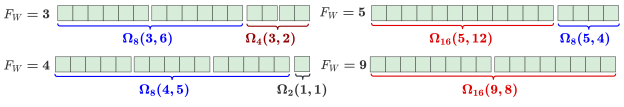

Figure 5: The fastest WinRS kernel pairs.

4.2 Baseline Segment Count Estimation

Given the fastest kernel pair, WinRS maximizes the number of segments to launch $\Omega_{\alpha_0}(n_0, r_0)$. As illustrated in Algorithm 1, WinRS estimates the baseline segment count $\widehat{Z}$ based on $\Omega_{\alpha_0}(n_0, r_0)$, while balancing a key trade-off: although raising $\widehat{Z}$ enhances parallelism, it causes extra partitioning overhead, including workspace and bucket-reduction time.

In a convolutional layer, FC, BDC, and BFC share similar computational complexity and the same parameter sets. The block counts

Algorithm 1 Simplified algorithm to estimate $\widehat{Z}$.

1: $\widehat{Z} \leftarrow (b_0 + b_1)/1.45b_2$
2: Calculate $\widehat{b_2}$ and Z_{max} based on N_{SM} and data size
3: **if** $\widehat{Z} < 2$ **and** $b_2 \geq \widehat{b_2}$**: return** $\widehat{Z} \leftarrow 1$
4: Calculate Z_1 based on computation intensity and N_{SM}
5: Calculate Z_2 based on time complexity
6: $\widehat{Z} \leftarrow \min(\widehat{Z},\ Z_1,\ Z_2,\ \frac{NO_H O_W}{512})$
7: $\widehat{Z} \leftarrow \min(P\lceil\frac{\widehat{Z}}{P}\rceil,\ Z_{max})$ # $P \leftarrow \min(2^{\lceil\log_2(\widehat{Z})\rceil},\ 8)$

for FC or BFC scale linearly with feature map dimensions, and is usually large due to high resolutions. Therefore, WinRS initializes $\widehat{Z}$ as $\frac{b_0+b_1}{1.45b_2}$ (**line 1**), where b_0, b_1, and b_2 are the block counts for FC, BDC, and BFC, respectively.

On a GPU, blocks are evenly distributed among N_{SM} streaming multiprocessors (SMs) for execution. To achieve full hardware utilization, the required block count scales proportionally with N_{SM}. Based on N_{SM} and the data size (total size of $\mathbf{X}$, $\nabla\mathbf{Y}$, $\nabla\mathbf{W}$), WinRS calculates two thresholds (**line 2**): $\widetilde{b_2}$, to check whether one segment can provide sufficient blocks for full SM utilization; and Z_{max}, the upper limit of $\widehat{Z}$ to avoid excessive overhead.

Since kernels with higher computation intensity can better hide latency, WinRS establishes a threshold k that is positively correlated with computation intensity. When $\widehat{Z} \geq kN_2/N_{SM}$, each SM has sufficient blocks to hide most latency. Beyond this threshold, raising $\widehat{Z}$ only marginally improves latency hiding, but linearly increases partitioning overhead. Based on N_{SM} and the computation intensity of $\Omega_{\alpha_0}(n_0, r_0)$, WinRS computes Z_1 to constrain $\widehat{Z}$ (**line 4**).

To make $\widehat{Z}$ positively correlated with the workload volume, WinRS calculates Z_2 based time complexity to limit $\widehat{Z}$ (**line 5**). $\widehat{Z}$ is padded to a multiple of 2, 4, or 8 for GPU friendliness, and finally constrained by Z_{max} (**line 7**).

4.3 Segment Shape Calculation

Given the baseline segment count $\widehat{Z}$, WinRS calculates the optimal segment height $\widehat{S_H}$ and width $\widehat{S_W}$, then partitions $\nabla\mathbf{Y}$ into $Z = \lfloor\frac{O_H}{\widehat{S_H}}\rfloor \times \lceil\frac{O_W}{\widehat{S_W}}\rceil$ segments, as shown in Figure 3. All segments, except those at the bottom and right edges, have a uniform shape of $\widehat{S_H} \times \widehat{S_W}$ to balance workload. As shown in Algorithm 2, WinRS calculates $\widehat{S_H}$ and $\widehat{S_W}$, subject to the constraint $Z \approx \widehat{Z} \leq \lfloor\frac{O_H}{\widehat{S_H}}\rfloor\lfloor\frac{O_W}{\widehat{S_W}}\rfloor$.

Algorithm 2 Simplified algorithm to calculate $\widehat{S_H}$ and $\widehat{S_W}$.

1: $\widehat{Z} \leftarrow \min(\widehat{Z},\ H_{max}W_{max})$
2: **if** $\widehat{Z} = 1$**:** $(\widehat{S_H},\ \widehat{S_W}) \leftarrow (O_H,\ r_0\lfloor\frac{O_W}{r_0}\rfloor)$ **terminate**
3: **if** $\widehat{Z} \geq W_{max}$**:** $(\widehat{S_H},\ \widehat{S_W}) \leftarrow (\lfloor\frac{O_H O_W}{\widehat{Z} r_0}\rfloor,\ r_0)$ **terminate**
4: **if** $W_{max}\%\ \widehat{Z} = 0$**:** $(\widehat{S_H},\ \widehat{S_W}) \leftarrow (O_H,\ r_0\lfloor\frac{W_{max}}{\widehat{Z}}\rfloor)$ **terminate**
5: $x \leftarrow$ min factor of W_{max} in $[\lfloor\frac{W_{max}}{\widehat{Z}}\rfloor,\ \lfloor\frac{H_{max}W_{max}}{\widehat{Z}}\rfloor]$
6: **if** x exists **:** $(\widehat{S_H},\ \widehat{S_W}) \leftarrow (\lfloor\frac{O_H O_W}{\widehat{Z} x r_0}\rfloor,\ x r_0)$ **terminate**
7: $(\widehat{S_H},\ \widehat{S_W}) \leftarrow (O_H,\ r_0\lfloor\frac{O_W}{r_0}\rfloor)$

Since $\widehat{S}_W$ should be a multiple of r_0 to match $\widehat{\Omega}_{\alpha_0}(n_0, r_0)$, and $\widehat{S}_H$ must exceed p_H to prevent zero segments; the maximum number of segments along height and width axes are $H_{max} = \lfloor \frac{O_H}{p_H} \rfloor$ and $W_{max} = \lceil \frac{O_W}{r_0} \rceil$, respectively (**line 1**).

$$Z = \lfloor \frac{O_H}{\widehat{S}_H} \rfloor (\lfloor \frac{O_W}{\widehat{S}_W} \rfloor + 1) \geq \widehat{Z}(1 + \lfloor \frac{O_W}{\widehat{S}_W} \rfloor^{-1}) \tag{5}$$

When O_W is not a multiple of $\widehat{S}_W$, inequality (5) holds and reveals that reducing $\widehat{S}_W$ decreases the total segment count Z, thus minimizing the computational redundancy in boundary segments. When $\widehat{Z} \geq W_{max}$, the minimum of $\widehat{S}_W$ is r_0 (**line 4**). Otherwise, there may exist integer x to let xr_0 be the minimum, where x must satisfy three constraints: (1) $\widehat{Z} \geq \lfloor \frac{O_W}{xr_0} \rfloor = \lfloor \frac{W_{max}}{x} \rfloor$; (2) $\widehat{Z} \leq H_{max} \lfloor \frac{W_{max} r_0}{xr_0} \rfloor$; (3) $W_{max} r_0 \,\%\, xr_0 = 0$. Therefore, x is the minimum factor of W_{max} within interval $[\lfloor \frac{W_{max}}{\widehat{Z}} \rfloor, \lfloor \frac{H_{max} W_{max}}{\widehat{Z}} \rfloor]$ (**line 6**).

5 Kernel Design

Although reduced-precision data types, like FP16 and INT8, can accelerate CNN inference, many training tasks still require FP32 for gradient calculation to avoid vanishing and exploding gradients. Therefore, WinRS starts from stable FP32, and then ported to FP16 with Tensor Core acceleration for enhanced throughput.

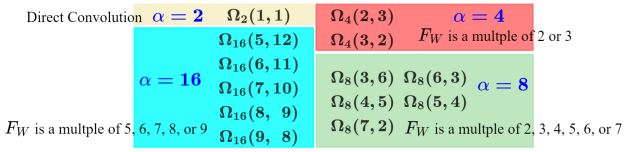

Figure 6: The 13 types of WinRS kernels.

As shown in Figure 6, WinRS kernels cover 13 distinct 1D Winograd convolutions, supporting $F_W \in \{nk | n \in \mathbb{N}^*, 2 \leq k \leq 9\}$. To balance throughput and numerical accuracy, the selected values of α are 2, 4, 8, and 16. Further, $\Omega_4(3,2)$, $\Omega_8(3,6)$, $\Omega_8(5,4)$, $\Omega_8(7,2)$, $\Omega_{16}(9,8)$, and $\Omega_{16}(7,10)$ have been ported from FP32 to FP16.

5.1 Common Execution Flow

Having established the method for workload distribution across segments, this section details the execution flow of $\Omega_\alpha(n,r)$ for the z-th segment $\nabla\mathbf{Y}(z) \in \mathbb{R}^{N \times S_H(z) \times S_W(z) \times O_C}$.

Given the $B_N \times B_M \times 8$ cache-block size[3], the workload on $\nabla\mathbf{Y}(z)$ is processed by a group of $\frac{O_C}{B_N} \times \frac{I_C}{B_M} \times \frac{F_H F_W}{n}$ blocks. Each block computes $B_N \times B_M$ output tiles, and is further divided into 8 warps, where each warp comprises 32 threads. The execution flow of a specific block is presented in Algorithm 3.

As formalized in (6), each output tile $\mathcal{Y} \subset \nabla\mathbf{W}(z)$ is computed by accumulating Winograd-convolution results between input tile $\mathcal{X}_{h,w} \subset \nabla\mathbf{X}$ and filter tile $\mathcal{W}_{h,w} \subset \nabla\mathbf{Y}(z)$. The left and right summations (Σ) represent Dimension Reduction and Kernel Split, respectively. By factoring out $\mathbf{A}^T$ from both summations, $\Omega_\alpha(n,r)$ combines all stages into the **main loop** (**line 23-28**), leaving only the OT to be executed afterwards. Dimension Reduction and Filter Split have minimal overhead, requiring only a few registers and contributing less than 1% of the total instructions.

[3]For FP32 kernels: the maximum $B_N \times B_M$ is 64×32 when $\alpha \in \{16, 8\}$, 64×64 when $\alpha = 4$, 128×128 when $\alpha = 2$. FP16 kernels utilize larger cache-block sizes: the maximum $B_N \times B_M$ is 64×64 when $\alpha = 16$, and 128×64 when $\alpha \in \{4, 8\}$.

Algorithm 3 Simplified execution flow of a $\Omega_\alpha(n,r)$ block. $\mathbf{A} \in \mathbb{R}^{\alpha \times n}$, $\mathbf{G} \in \mathbb{R}^{\alpha \times r}$, and $\mathbf{D} \in \mathbb{R}^{\alpha \times \alpha}$ are transform matrices of $F(n,r)$; z is the segment index, and b_z is block index.

1: **Function** `fetch_data`($\nabla\mathbf{Y}$, $\mathbf{X}$) **:**
2: $\mathcal{W}[B_N][8][r] \leftarrow B_N \times 8$ filter tiles $\subset \nabla\mathbf{Y}$
3: $\hat{\mathcal{W}}[B_N][8][\alpha] \leftarrow \mathbf{G}\mathcal{W}$ # filter transform
4: $\mathcal{X}[B_M][8][\alpha] \leftarrow B_M \times 8$ input tiles $\subset \mathbf{X}$
5: $\hat{\mathcal{X}}[B_M][8][\alpha] \leftarrow \mathbf{D}^T\mathcal{X}$ # input transform
6: $\mathbf{Gs}[buf][][0:w_i][0:\frac{B_N}{32}l_i] \leftarrow \hat{\mathcal{W}}[0:w_i][0:\frac{B_N}{32}l_i][]$
7: $\mathbf{Ds}[buf][][0:w_i][0:\frac{B_M}{32}l_i] \leftarrow \hat{\mathcal{X}}[0:w_i][0:\frac{B_M}{32}l_i][]$
8: **syncthreads**
9: **Function** `batched_GEMM`($\mathbf{v}$, $\mathbf{Gs}$, $\mathbf{Ds}$)**:**
10: $\mathbf{v}[0:\alpha]$ += $\mathbf{Gs}[0:\alpha]^T\mathbf{Ds}[0:\alpha]$ # α-batched GEMM
11: $buf \leftarrow (buf+1) \bmod N_{buf}$ # switch buffer
12: **Function** `write_result`($\mathbf{v}$, $\nabla\widehat{\mathbf{W}}$)**:**
13: **SMEM** $\mathbf{As}[\alpha][\frac{B_N B_M}{R}]$ # reuse the space of **Gs** and **Ds**
14: **for** $i \leftarrow 0$ **to** R **:**
15: **syncthreads**; $\mathbf{As} \leftarrow \frac{1}{R}$ elements of $\mathbf{v}$
16: **syncthreads**; $\hat{\mathcal{Y}}[\frac{B_N B_M}{R}][\alpha] \leftarrow \mathbf{As}^T$ # SMEM to registers
17: $\mathcal{Y} \leftarrow \mathbf{A}^T\hat{\mathcal{Y}}$ # output transform
18: write $\mathcal{Y}$ to the z-th $\nabla\widehat{\mathbf{W}}$ bucket
19:
20: $\mathbf{v}[\alpha][B_N][B_M] \leftarrow \mathbf{0}$ # accumulators
21: $buf \leftarrow 0$ # buffer index
22: **SMEM** $\mathbf{Gs}[N_{buf}][\alpha][8][B_N]$, $\mathbf{Ds}[N_{buf}][\alpha][8][B_M]$
23: **for** $k \leftarrow 0$ **to** $S_H(z,b_i)S_W(z)$ **by** r**:**
24: `fetch_data`($\nabla\mathbf{Y}$, $\mathbf{X}$)
25: **for** $n \leftarrow 8$ **to** N by 8 **:**
26: `batched_GEMM`($\mathbf{v}$, $\mathbf{Gs}$, $\mathbf{Ds}$)
27: `fetch_data`($\nabla\mathbf{Y}$, $\mathbf{X}$)
28: `batched_GEMM`($\mathbf{v}$, $\mathbf{Gs}$, $\mathbf{Ds}$)
29: `write_result`($\mathbf{v}$, $\nabla\widehat{\mathbf{W}}$)

$$\mathcal{Y} = \sum_{h=1}^{S_H(z)S_W(z)/r} \sum_{w=1} \mathbf{A}^T[(\mathbf{G}\mathcal{W}_{h,w}) \odot (\mathbf{D}^T\mathcal{X}_{h,w})] \tag{6}$$

Data Fetching (line 1-9). In every iteration of the main loop, a block loads $B_N \times 8$ filter tiles and $B_M \times 8$ input tiles. As tensors are organized in NHWC layout, each warp's 32 threads load continuous B_N and B_M tiles along I_C and O_C axes, using vectorized operations to enhance memory bandwidth. The loaded tiles are transformed and stored to SMEM at continuous offsets, with each thread's store location determined by **warp index** w_i and **lane index** l_i.

Within a block, since all threads load data from the same height-axis locations, the data-loading regions are clipped to bypass height-axis zero padding of size p_H. As shown in Figure 7, given block index b_i, the **X**-loading area in is clipped to height-axis coordinates between ih_s and ih_e. Accordingly, the height of $\nabla\mathbf{Y}(z)$-loading area

is clipped to $S_H(z, b_i) = ih_e - ih_s$ (**line 23**). This optimization reduces time complexity by $\frac{p_H(p_H+1)}{F_H O_H}$, and can avoid more redundant calculations with small feature maps.

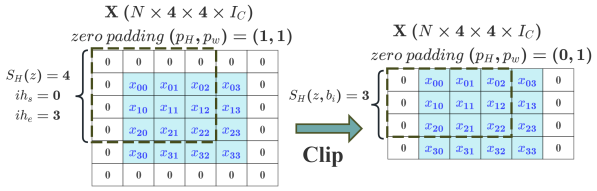

Figure 7: Data-loading area clipping. With a zero padding of 1, the data-loading area is clipped from 6×6 to 4×6, reducing time complexity by 12.5%.

Width-axis zero padding of size p_W is implicitly handled. For FP32 kernels, addresses are masked to -1, making texture memory return zero automatically. For FP16 kernels, conditional statements verify address boundaries instead. As $r \times r$ filters are commonly used with $\lfloor \frac{r}{2} \rfloor$ padding, certain kernels are optimized for $p_W \leq \lfloor \frac{r}{2} \rfloor$ to further simplify zero-padding process.

Batched GEMM (line 9-11). The transformed $B_N \times 8$ filter tiles and $B_M \times 8$ input tiles are derived from independent input and output channels. To maximize computation intensity, their corresponding EWMs are converted into an α-batched $B_N \times B_M \times 8$ general matrix multiplication (GEMM).

FP32 kernels execute the batched GEMM on CUDA Cores, assigning each GEMM to a group of $\frac{256}{\alpha}$ threads. Each GEMM proceeds through 8 rounds of $B_N \times B_M$ outer products, where the outer products are decomposed into $m\times8$ sub-operations for thread parallelization ($m \in \{8, 16\}$). Given thread-specific offsets u_x K_i, G_i, and D_i, the execution flow of the thread is as follows:

1: **for** $k \leftarrow 0$ **to** 8 **:**
2: $\quad$ $\mathbf{a}[m] \leftarrow \mathbf{Gs}[buf][K_i][k][G_i : G_i + m]$ # SMEM to registers
3: $\quad$ $\mathbf{b}[8\ ] \leftarrow \mathbf{Ds}[buf][K_i][k][D_i : D_i + 8\ ]$ # SMEM to registers
4: $\quad$ $\mathbf{v}[u_x][G_i : G_i + m][D_i : D_i + 8] \mathrel{+}= \mathbf{a} \otimes \mathbf{b}$

FP16 kernels accelerate the batched GEMM with Tensor Cores, distributing $\frac{1}{8}$ computation per warp. Each GEMM is decomposed into $16 \times 8 \times 8$ sub-operations and implemented via PTX assembly. Key PTX instructions include: (1) *ldmatrix.sync.aligned.x4.trans.m8n8*, loading four 8×8 matrix tiles from SMEM with automatic transposition; (2) *mma.sync.aligned.m16n8k8*, performing a $16 \times 8 \times 8$ GEMM with warp-level synchronization. The execution flow of the w_i-th warp is as follows ($\alpha \geq 8$):

1: **for** $\beta \leftarrow w_i \frac{\alpha}{8}$ **to** $(w_i + 1)\frac{\alpha}{8}$ **:**
2: **for** $i \leftarrow 0$ **to** B_N **by** 32**:**
3: **for** $j \leftarrow 0$ **to** B_M **by** 32**:**
4: $\quad$ $\mathbf{A}_i[32][8] \leftarrow \mathbf{Gs}[buf][\beta][0:8][i : i + 32]$ #ldmatrix.x4
5: $\quad$ $\mathbf{B}_j[32][8] \leftarrow \mathbf{Ds}[buf][\beta][0:8][j : j + 32]$ #ldmatrix.x4
6: $\quad$ $\mathbf{v}[\beta][i : i + 32][j : j + 32] \mathrel{+}= \mathbf{A}_i\mathbf{B}_j^T$ # 2×4 mma.m16n8k8

Result Output (line 13-18). After the main-loop, each block retains $B_N \times B_M$ pre-transformed output tiles in accumulators, with tile elements distributed across α threads. Due to SMEM-capacity constraints, OT is completed over R rounds ($R \in \{4, 8\}$). In each round, threads store $\frac{1}{R}$ of accumulators to SMEM array **As**; then the transposed **As** data is loaded into registers to perform OT. To enhance bandwidth, output tiles are written to global memory via wide memory transactions.

5.2 Optimization Techniques

Software Pipelining. Upon completing the batched GEMM, threads prefetch and transform tiles for the next iteration. Across different warps, the batched GEMM, prefetching, and transforms are concurrently executing on SMs, which overlaps computations with memory access latency. SMEM is doubled buffered ($N_{buf} = 2$) to further enhance warp parallelism and optimize throughput.

Bank Conflict Elimination. Threads within a warp may compete to access the same SMEM bank, leading to bank conflicts.

In FP16 Batched GEMM, to prevent bank conflicts from *ldmatrix* instructions, the last dimensions of SMEM array **Gs** and **Ds** are padded with 128 bits, yielding 46% throughput improvement.

In FP32 Batched GEMM, to avoid bank conflicts from 128-bit SMEM loading, thread-specific offsets G_i and D_i are contrapuntally arranged, enhancing throughput by about 5%:

1: $u_y,\ u_x \leftarrow \lfloor T_i/\alpha \rfloor,\ T_i \bmod \alpha\ \#\ T_i \leftarrow 32w_i + l_i$
2: $G_i,\ D_i \leftarrow 8(2\lfloor \frac{u_y}{\theta} \rfloor + u_y \bmod 2),\ 8\lfloor \frac{u_y \bmod \theta}{2} \rfloor\ \#\ \theta\ \leftarrow \frac{B_M}{8}$

In Result Output, to eliminate bank conflicts from SMEM storing, the last dimension of **As** is padded with 128 bits for FP32 and 64 bits for FP16, improving throughput by about 2%.

Transform Simplification. The low-rank transform matrices **A**, **G**, and **D** exhibit structured symmetry, when calculated using interpolation points $\in \{0, \pm1, \pm2, \pm\frac{1}{2}, \pm3, \pm\frac{1}{3}...\}$. As shown Figure 8, the $2k$-th and $(2k+1)$-th ($k \geq 0$) rows of **A**, **G** and $\mathbf{D}^T$ have equivalent and opposite elements in even and odd positions, respectively. For IT, FT and OT, this property enables the reuse of multiplication results, which nearly halves the required multiplications and improves throughput by about 6%.

$$A = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 1 & -1 & 1 \\ 1 & 2 & 4 \\ 1 & -2 & 4 \\ 1 & \frac{1}{2} & \frac{1}{4} \\ 1 & -\frac{1}{2} & \frac{1}{4} \\ 0 & 0 & 0 \end{bmatrix} \quad G = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ -\frac{2}{9} & -\frac{2}{9} & -\frac{2}{9} & -\frac{2}{9} & -\frac{2}{9} & -\frac{2}{9} \\ -\frac{2}{9} & \frac{2}{9} & -\frac{2}{9} & \frac{2}{9} & -\frac{2}{9} & \frac{2}{9} \\ \frac{1}{90} & \frac{2}{90} & \frac{4}{90} & \frac{8}{90} & \frac{16}{90} & \frac{32}{90} \\ \frac{1}{90} & -\frac{2}{90} & \frac{4}{90} & -\frac{8}{90} & \frac{16}{90} & -\frac{32}{90} \\ \frac{64}{90} & \frac{32}{90} & \frac{16}{90} & \frac{8}{90} & \frac{4}{90} & \frac{2}{90} \\ \frac{64}{90} & -\frac{32}{90} & \frac{16}{90} & -\frac{8}{90} & \frac{4}{90} & -\frac{2}{90} \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad D^T = \begin{bmatrix} 1 & 0 & -\frac{21}{4} & 0 & \frac{21}{4} & 0 & -1 & 0 \\ 0 & 1 & 1 & -\frac{17}{4} & -\frac{17}{4} & 1 & 1 & 0 \\ 0 & -1 & 1 & \frac{17}{4} & -\frac{17}{4} & -1 & 1 & 0 \\ 0 & \frac{1}{2} & \frac{1}{4} & -\frac{5}{2} & -\frac{5}{4} & 2 & 1 & 0 \\ 0 & -\frac{1}{2} & \frac{1}{4} & \frac{5}{2} & -\frac{5}{4} & -2 & 1 & 0 \\ 0 & 2 & 4 & -\frac{5}{2} & -5 & \frac{1}{2} & 1 & 0 \\ 0 & -2 & 4 & \frac{5}{2} & -5 & -\frac{1}{2} & 1 & 0 \\ 0 & -1 & 0 & \frac{21}{4} & 0 & -\frac{21}{4} & 0 & 1 \end{bmatrix}$$

Figure 8: Transform matrices of Winograd $F(3, 6)$.

Accuracy Optimization. FP16 kernels incorporate certain FP32 operations, accounting for 0.6% to 18% of FP16 operations. FT, IT, and OT are temporarily computed in FP32 precision, and finally converted to FP16. WinRS partitions convolution accumulation into multiple $\nabla\mathbf{Y}$ segments, mitigating numerical overflow. The reduction kernel sums all partition results using FP32 Kahan summation, to minimize accuracy loss.

The limited dynamic range of FP16 poses challenges for $\Omega_{16}(n, r)$ transform matrices, whose elements span magnitude from 10^{-8} to 10^5. However, the row-wise magnitude coherence enables the incorporation of scaling matrices $\mathbf{A}_s$, $\mathbf{G}_s$, and $\mathbf{D}_s$, as shown in (7). $\mathbf{G}_s$ and $\mathbf{D}_s$ normalize row elements in **G** and **D**, enforcing unit L1-norm per row to minimize changes to data magnitude. In OT,

$\mathbf{A}_s$ rescales accumulators to correct values after FP32 conversion, leveraging its wider dynamic range to prevent numerical overflow.

$$\mathcal{Y} = (\mathbf{A}_s\mathbf{A})^T[((\mathbf{G}_s\mathbf{G})\mathcal{W}) \odot ((\mathbf{D}_s\mathbf{D})^T\mathcal{X})] \quad (7)$$

6 Evaluation

We evaluate WinRS in terms of workspace, throughput, and accuracy, against the high-performance cuDNN-9.8 library.

Benchmarks include five cuDNN BFC algorithms: (1) Cu-GEMM, encompassing GEMM-based Cu-Algo0, Cu-Algo1, and Cu-Algo3; (2) Cu-FFT, FFT-based; (3) Cu-WinNF, cuDNN's sole Winograd BFC, which is non-fused and supports 3×3 and 5×5 $\nabla\mathbf{W}$. Notably, only Cu-Algo1 and Cu-WinNF support FP16 Tensor-Core acceleration, with FP16 Cu-WinNF limited to 3×3 $\nabla\mathbf{W}$.

The FP32 test is on RTX 4090 and RTX 3090 (flagship consumer GPUs, 24 GB). The FP16 test is on RTX 4090, L40S (data-center GPU, 48GB), and RTX A5000 (workstation GPU, 24 GB).

BFC parameters are based on common CNN architectures: (1) $\nabla\mathbf{W}$ shape $F_H\times F_W$ is from 2×2 to 9×9; (2) channel sizes I_C and O_C are 64, 128, 256, 512, or 1024, with $I_C = O_C$; (3) feature map shapes $O_H\times O_W$ and $I_H\times I_W$ are factors of standard resolutions $\in\{400\times400, 384\times384, 224\times224, 128\times128\}$, or multiples of r to only test $\Omega_\alpha(n,r)$; (4) batch size N is 32, 64, 128, or 256; (5) channel sizes are doubled when feature-map shapes are halved, to ensure consistent time complexity.

6.1 Workspace Evaluation

The experimental data size is from 52 MB to 2032 MB. For reference, a single convolutional layer in VGG [27] consumes up tp 383.7 MB during inference, with 32 batch size and 224×224 resolution. Table 2 summarizes the workspace of algorithms, except for Cu-Algo0 requiring no workspace.

Table 2: Algorithm workspace. the data marked × indicates the multiple of workspace relative to data size.

Algorithm	Average		Min		Max	
WinRS	37.9 MB	0.18×	0.0 MB	0.00×	134.8 MB	1.67×
Cu-Algo1	358.3 MB	1.06×	36.3 MB	0.28×	2035.7 MB	2.21×
Cu-Algo3	17.7 MB	0.10×	0.0 MB	0.00×	324.6 MB	0.91×
Cu-FFT	2948.0 MB	9.09×	584.5 MB	3.11×	17536 MB	30.40×
Cu-WinNF	956.0 MB	2.67×	180 MB	2.23×	2593.6 MB	5.90×

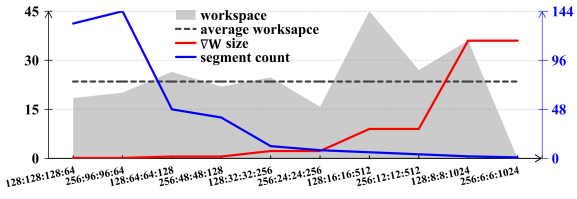

Figure 9: WinRS workspace for 3×3 $\nabla\mathbf{W}$ on RTX 4090. Left y-axis: workspace and $\nabla\mathbf{W}$ size in MB. Right y-axis: $\nabla\mathbf{Y}$ segment count. X-axis: $\nabla\mathbf{Y}$ dimensions in N:O_H:O_W:O_C format.

WinRS workspace ranges from 0 MB to 134.8 MB, with an average of 37.9 MB, which corresponds to 0× to 1.67× the data size, averaging 0.18×. Compared to Cu-Algo1, Cu-FFT, Cu-WinNF, WinRS utilizes only 10.6%, 1.29%, and 3.96% of their respective average workspace (348.3 MB, 2948 MB, 956 MB). While WinRS requires slightly more workspace than Cu-Algo3 (17.7 MB on average), it achieves demonstrably higher throughput.

WinRS workspace remains small across all channel sizes, as shown in Figure 9. When channel size is small (e.g. 64, 128), a relatively large number of $\nabla\mathbf{Y}$ segments are generated to enhance parallelism; however, the workspace is small due to the tiny size of $\nabla\mathbf{W}$. As channel sizes increase (e.g. 256, 512), each $\nabla\mathbf{Y}$ segment provides more blocks, thus reducing the segment count. When channel sizes are sufficiently large (e.g. 1024), a single $\nabla\mathbf{Y}$ segment provides sufficient blocks, resulting in 0 workspace.

6.2 Throughput Evaluation

In experiments, the time complexity ranges from 137 to 2174 GFLOPs, sufficiently large to maximize GPU utilization. Throughput is calculated as $\frac{2O_CF_HF_WI_CO_HO_WN}{\hat{t}}$, where the execution time $\hat{t}$ is estimated by averaging 1000 repeated runs. Cu-GEMM represents the fastest algorithm among Cu-Algo0, Cu-Algo1, and Cu-Algo3. Figure 10 and 11 show algorithm throughput, and the speedup of WinRS over cuDNN is summarized in Table 3.

Table 3: WinRS speedup over cuDNN. Each cell displays the speedup data in a format of '*average*: *minimum-maximum*'.

$F_H\times F_W$	FP32: RTX 4090			FP32: RTX 3090		
	Cu-GEMM	Cu-FFT	Cu-WinNF	Cu-GEMM	Cu-FFT	Cu-WinNF
2×2	1.51: 1.05-2.00	7.85: 2.20-16.8	N/A	1.35: 1.06-1.63	5.82: 1.38-12.6	N/A
3×3	1.95: 1.38-2.79	5.87: 1.29-17.3	2.24: 0.64-6.33	1.53: 1.21-2.23	3.92: 0.96-8.15	1.39: 0.62-3.38
4×4	1.59: 1.26-2.33	3.91: 1.16-11.0	N/A	1.59: 1.27-1.97	2.58: 0.83-5.08	N/A
5×5	2.17: 1.64-2.69	3.48: 0.99-10.1	1.21: 0.51-2.68	2.05: 1.66-2.75	2.00: 0.66-4.36	0.87: 0.46-1.78
6×6	2.43: 1.68-3.56	2.82: 0.99-6.60		1.86: 1.59-2.26	1.63: 0.63-3.08	
7×7	2.41: 1.98-2.96	2.73: 0.89-7.16	N/A	2.23: 1.44-3.20	1.41: 0.62-2.76	N/A
8×8	2.38: 1.92-3.17	2.65: 0.88-5.95		2.17: 1.82-2.63	1.44: 0.59-2.72	
9×9	2.52: 1.83-2.95	2.38: 0.99-5.08		2.16: 1.77-2.62	1.33: 0.55-2.23	

$F_H\times F_W$	FP16: RTX 4090		FP16: L40S		FP16: RTX A5000	
	Cu-GEMM	Cu-WinNF	Cu-GEMM	Cu-WinNF	Cu-GEMM	Cu-WinNF
3×3	2.21: 1.54-2.96	2.17: 0.86-3.80	2.43: 1.43-4.70	2.52: 0.84-4.77	1.81: 1.32-2.25	1.20: 0.58-2.04
5×5	2.22: 1.61-2.90		2.03: 1.84-2.23		1.96: 1.59-2.98	
7×7	2.12: 1.60-2.57	N/A	1.97: 1.55-2.46	N/A	1.80: 2.10-1.59	N/A
9×9	2.41: 1.90-2.96		2.04: 2.59-1.66		2.08: 1.73-2.63	

For FP32, WinRS speedup is 1.05× to 3.56× over Cu-GEMM, 0.55× to 17.3× over Cu-FFT, and 0.46× to 6.33× over Cu-WinNF. For FP16, WinRS speedup is 1.32× to 4.7× over Cu-GEMM, and 0.58× to 4.77× over Cu-WinNF. On average, WinRS's FP16 Tensor-Core implementations achieve 3.27× speedup over its FP32 CUDA-Core baseline. Due to reduced time complexity, WinRS, Cu-FFT, and Cu-WinNF can exceed the theoretical throughput of hardware.

WinRS has higher throughput under following conditions: (1) larger F_W promotes time-complexity reduction through larger transform matrices; (2) increased channel sizes lead to more blocks per $\nabla\mathbf{Y}$ segment, which enhances parallelism and reduces bucket-reduction overhead; (3) smaller feature-map sizes allow the clip of more height-axis zero padding, where the smaller data size also improves the cache hit ratio.

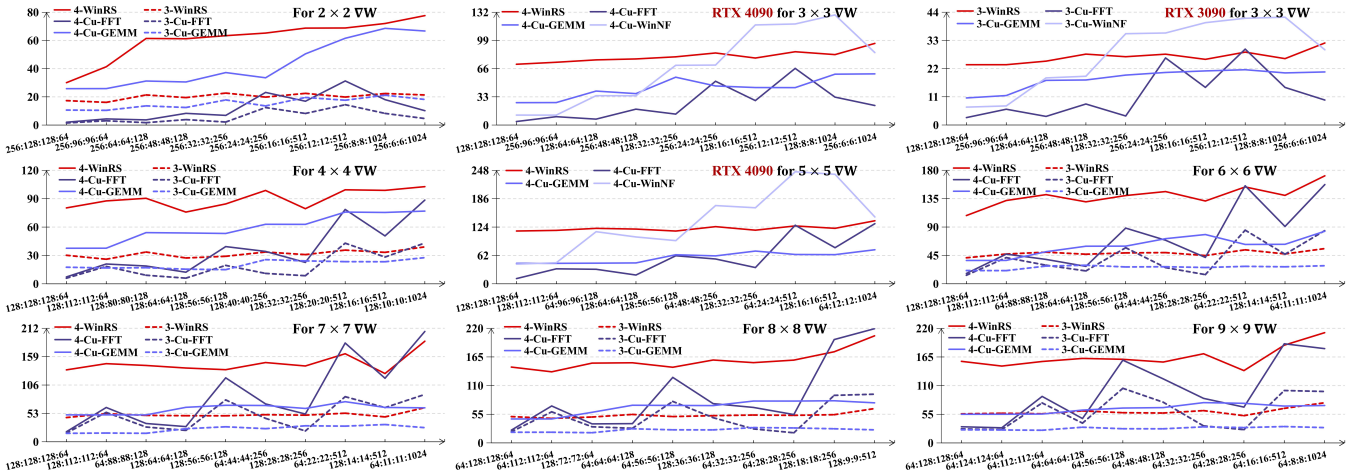

Figure 10: FP32 throughput on RTX 4090 and RTX 3090. Throughput (y-axis) is in TFLOPS. ∇Y dimensions (x-axis, N:O_H:O_W:O_C format) maintain comparable time complexity in each sub-figure. Prefix 4 and 3 denote RTX 4090 and 3090, respectively.

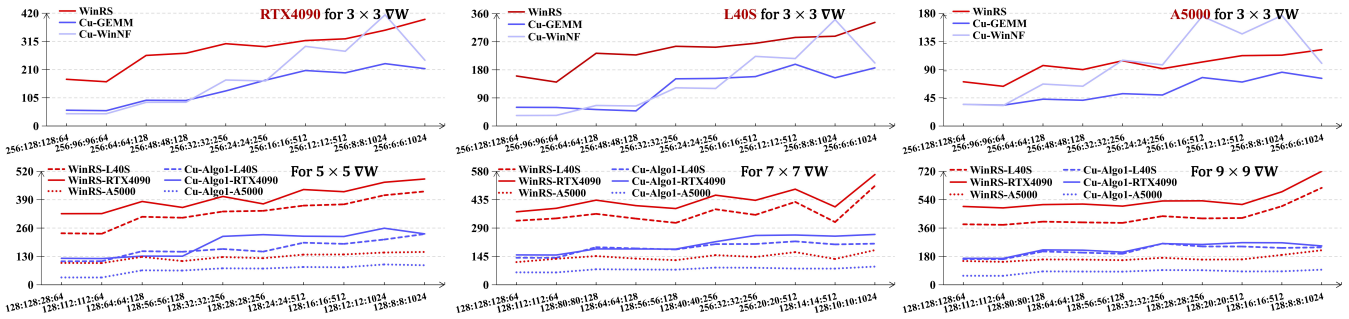

Figure 11: FP16 throughput on L40S, RTX 4090, and RTX A5000. Throughput (y-axis) is in TFLOPS. ∇Y dimensions (x-axis, N:O_H:O_W:O_C format) maintain comparable time complexity in each sub-figure.

Compared to Cu-FFT and Cu-WinNF, WinRS has asymptotic time complexity but larger constant factors[4]. However, WinRS can outperform them for two advantages: (1) Cu-FFT and Cu-WinNF execute IT, FT, OT, and EWM in four separate kernels, with global-memory workspace at least 2.23× the data size, incurring substantial I/O overhead for intermediate data movement. WinRS stores intermediate results in on-chip SMEM, with 10× higher bandwidth than global memory, significantly reducing I/O time. (2) Although the EWM stages of Cu-FFT and WinNF have higher computation intensity than WinRS, they cannot run in parallel with IT, FT, and OT. As a result, the dense EWM computations can not overlap the memory latency of IT, FT, and OT via software pipelining. Conversely, WinRS kernels execute all stages concurrently through full fusion, enhancing latency hiding and hardware utilization.

For an algorithm, the throughput is crucially affected by its hardware-resource demands. A breakdown is given in (8), where: $\hat{T}$ is the expected execution time; C_{time} is the time complexity; C_{data} is I/O volume of moving intermediate results; V_{comp} and V_{band} are hardware's computing capability and bandwidth, respectively.

$$\hat{T} = \frac{C_{\text{time}}}{V_{\text{comp}}} + \frac{C_{\text{data}}}{V_{\text{band}}} \tag{8}$$

Fused-Algorithms (e.g. WinRS and Cu-GEMM) are primarily governed by $\frac{C_{\text{time}}}{V_{\text{comp}}}$, due to minimal intermediate-result I/O. In contrast, Non-fused Algorithms (e.g. Cu-FFT and Cu-WinNF) are additionally influenced by $\frac{C_{\text{data}}}{V_{\text{band}}}$. Here are two key observations:

Observation 1. Non-fused algorithms are more sensitive to tensor dimensions. With a fixed product of channel sizes and feature-map shapes, C_{time} remains invariant, but C_{data} increases with smaller channels and larger features, leading to higher I/O overhead and therefore lower throughput. In contrast, fused-algorithm throughput are more robust across varying tensor dimensions.

Observation 2. Fused algorithms favour computing capability over memory bandwidth. While the throughput of fused algorithms scales almost linearly with V_{comp}, non-fused algorithms are limited by a weighting of V_{comp} and V_{band}. From RTX 3090 to RTX 4090, V_{comp} and V_{band} increase by 132% and 8%; from FP32 CUDA Cores to FP16 Tensor Cores, V_{comp} and V_{band} increase by 297% and 100%. As a result, WinRS has higher throughput relative to non-fused algorithms on RTX 4090 than RTX 3090, and in FP16 than FP32.

[4]WinRS reduces time complexity by 1.5× to 4.5×. Cu-FFT reduces time complexity by $\gamma F_H F_W$ ×, where γ is a constant. For 3×3 and 5×5 **∇W**, Cu-WinNF reduces time complexity by 4× and 6.25×, whereas the reduction of WinRS is 2.25× and 3.75×.

Specifically, for 3×3 $\nabla\mathbf{W}$, FP16 WinRS outperforms Cu-WinNF when $O_C \leq 512$ on RTX 4090, whereas FP32 WinRS is faster at $O_C \leq 256$ and $O_C \leq 128$ on RTX 3090. L40S achieves similar FP16 throughput to RTX4090, due to its comparable V_{comp} and V_{band}. Compared to RTX 4090, RTX A5000 has a lower ratio of V_{comp} to V_{band}. This makes non-fused algorithms perform better, where FP16 WinRS outperforms Cu-WinNF for $O_C \leq 256$.

Generally, WinRS has higher throughput than Cu-WinNF and Cu-FFT, in memory-bound and compute-intensive scenarios.

6.3 Accuracy Evaluation

Accuracy is evaluated using Mean Absolute Relative Error (MARE), against FP64 ground truth. Tensors are generalized with uniform distribution in $[0, 1]$. For FP16 tests, $\nabla\mathbf{Y}$ is scaled by 10^{-2} to prevent numerical overflow. Table 4 summarizes the MAREs of algorithms.

Table 4: The MAREs of algorithms.

Algorithm	FP32: min	FP32: max	FP16:min	FP16: max
WinRS $\Omega_4(n, r)$	1.16e-7	4.79e-7		
WinRS $\Omega_8(n, r)$	1.13e-7	8.26e-7	3.35e-4	2.69e-3
WinRS $\Omega_{16}(n, r)$	9.52e-6	1.34e-5	8.75e-4	1.09e-2
Cu-FFT	7.23e-8	1.45e-7		
Cu-Algo0/Algo3	7.01e-8	5.92e-7		
Cu-WinNF	4.78e-7	3.68e-6	1.59e-3	6.52e-1
Cu-Algo1	4.64e-5	1.78e-3	5.69e-4	8.34e-1

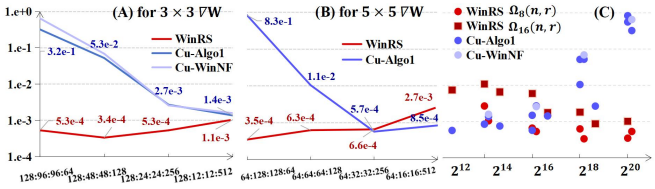

Figure 12: FP16 MARE distribution. (A, B): MARE versus $\nabla\mathbf{Y}$ dimensions in N:O_H:O_W:O_C format. (C): MARE versus accumulation length NO_HO_W.

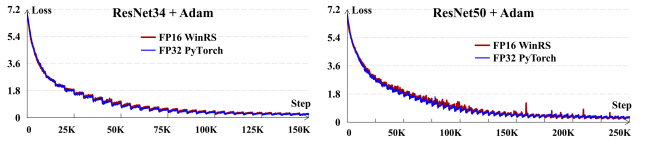

Figure 13: CNN training loss on ImageNet-1K.

For FP32, $\Omega_4(n, r)$ and $\Omega_8(n, r)$ achieve MARE $\sim 10^{-7}$, slightly less accurate than Cu-Algo0 but more accurate than Cu-WinNF. The MARE of $\Omega_{16}(n, r)$ is around 10^{-5}, less accurate than Cu-WinNF but more accurate than Cu-Algo1.

For FP16, $\Omega_8(n, r)$ achieves MAREs $\sim 10^{-3}$ to 10^{-4}, slightly outperforming Cu-Algo1. $\Omega_{16}(n, r)$ has an accuracy around 10^{-3}, between Cu-Algo1 and Cu-WinNF. As Figure 12 shows, when the accumulation length is large ($NO_HO_W \geq 2^{18}$, typical in early CNN layers), Cu-Algo1 and Cu-WinNF can degrade in accuracy, whereas WinRS maintains accuracy through accumulation segmentation and FP32 Kahan summation. With smaller accumulation lengths (common in deeper layers), WinRS is slightly less accurate due to fewer $\nabla\mathbf{Y}$ segments, but still comparable to Cu-Algo1.

We used FP32 WinRS to train VGG16, VGG19 [27], ResNet34, and ResNet50 [28] on ImageNet-1K [31] and Cifar10 [32] datasets. The resulting accuracy is similar to PyTorch, with differences less than ±0.6% across all models. FP16 WinRS was used to train ResNet34 and ResNet50 on ImageNet-1K with Loss Scaling technique, achieving similar convergence to FP32 PyTorch, as shown in Figure 13.

6.4 Performance Summary

Prior evaluations have demonstrated WinRS's high throughput and memory efficiency. In terms of flexibility, WinRS offers a much wider applicable range than Cu-WinNF. Compared to Cu-FFT's universal coverage, although WinRS supports a slightly narrower range of $F_H \times F_W$, it consistently outperforms Cu-GEMM, whereas Cu-FFT lags behind Cu-GEMM with small $F_H \times F_W$. These complementary strengths enable WinRS to achieve comparable real-world applicability to Cu-FFT. Thus, WinRS is a fast, memory-efficient, and flexible BFC algorithm.

7 Related Works

Winograd Implementations. Winograd convolution has been implemented on GPUs [3–8, 14–16, 25], CPUs [9–11, 17], and FPGAs [16, 18–24, 35]. Most [3–6, 8–11, 15, 17–19] target FC or BDC with small filters (typically 3×3). Although many GPU implementations are based on CUDA Cores, a few non-fused-Winograd approaches [5, 7] exploit Tensor Cores.

WinRS is a fused-Winograd convolution with Tensor-Core acceleration. Combined with dimension reduction and filter split, WinRS leverages a wider variety of Winograd convolutions to enhance flexibility. While cuDNN [5] provides efficient Winograd BFC, WinRS features full kernel fusion and adaptive workload distribution, leading to better performance in certain cases.

Most methods rely on 2D Winograd convolution, but WinRS and Im2col-Winograd [14] employ the 1D approach. However, Im2col-Winograd is designed for filter widths from 2 to 9, and uses fixed workload distribution, limiting its applicability to BFC; besides, its limitation to FP32 CUDA Cores restricts its throughput.

Decomposition Winograd. Previous studies [14, 16, 20–25, 36] extend Winograd convolution to larger filters through decomposition, but only DWM [25] and WinTA [23] support BFC. WinTA is an FPGA implementation. DWM presents a mathematically elegant approach for BFC, but detailed information of its GPU design is limited. While these works mainly focus on large filters, WinRS is also optimized for small outputs via adaptive workload distribution, crucial for BFC performance as it ensures sufficient parallelism and minimized partitioning overhead.

A limitation in DWM, WinTA, and other methods [16, 21–25, 36] is their multi-stage execution of Winograd convolution and decomposition, leading to increased workspace for intermediate results and data movement between processing units. WinRS tackles this problem by fusing all stages into one.

The nested decomposition [24] requires both large filters and outputs to recursively reduce time complexity, conflicting with the

small-output characteristic of BFC. Moreover, its recursive nature hiders kernel fusion, a key optimization in WinRS.

8 Conclusion

This work proposes WinRS, a fast, memory-efficient, and flexible BFC algorithm. Through dimension Reduction and filter Split, WinRS matches N-D large filters with the fastest kernels, and achieves full kernel fusion to mitigate memory bottlenecks. Based on adaptive workload distribution, WinRS balances computations across an optimal number of blocks, effectively optimizing hardware utilization in small-output cases. Furthermore, WinRS leverages 13 distinct Winograd convolutions and Tensor-Core acceleration, enhancing both flexibility and throughput.

WinRS has higher throughput for larger filter gradients, aligning with the current trend towards larger filters [30, 37, 38]. FP16 WinRS kernels can be ported to BF16, and further to FP8 and INT8. With moderate modifications, WinRS can support FC and BDC. WinRS is open-source and adaptable for future use.

Acknowledgments

We sincerely thank the reviewers for their constructive comments. This work was supported by national key R&D program of China (No. 2024YFD1200100), key science & technology project of Anhui province (No. 202423m10050002), and Anhui provincial key research and development program (No. 2023n06020016, 2023n06020028).

References

[1] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, and et al. cuDNN: Efficient Primitives for Deep Learning. *CoRR*, abs/1410.0759, 2014, 2014.
[2] Yangqing Jia, Evan Shelhamer, Jeff Donahue, and et al. caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia Retrieval (ICMR)*, pages 675–678, 2014.
[3] Andrew Lavin and Scott Gray. Fast algorithms for convolutional neural networks. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 4013–4021, 2016.
[4] Roberto L. Castro, Diego Andrade, and Basilio B. Fraguela. OpenCNN: A Winograd Minimal Filtering Algorithm Implementation in CUDA. *Mathematics*, 9(17), 2021.
[5] NVIDIA. cuDNN, 2025. https://developer.nvidia.com/cudnn.
[6] Da Yan, Wei Wang, and Xiaowen Chu. Optimizing Batched Winograd Convolution on GPUs. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 32–44. ACM, 2020.
[7] Junhong Liu, Dongxu Yang, and Junjie Lai. Optimizing Winograd-Based Convolution with Tensor Cores. In *Proceedings of the 50th International Conference on Parallel Processing (ICPP)*, pages 1–10. ACM, 2021.
[8] Guangli Li, Lei Liu, Xueying Wang, Xiu Ma, and Xiaobing Feng. 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)*, pages 3842–3846. IEEE, 2020.
[9] Ruofan Wu, Feng Zhang, Jiawei Guan, and et al. DREW: Efficient Winograd CNN Inference with Deep Reuse. In *Proceedings of the ACM Web Conference 2022April 2022*, pages 1807–1816. ACM, 2021.
[10] Partha Maji, Andrew Mundy, and Ganesh et al. Dasika. 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)*, pages 1–5, 2019.
[11] Dosheng Li, Dan Huang, Zhiguang Chen, and Yutong Lu. Optimizing massively parallel winograd convolution on arm processor. In *Proceedings of the 50th International Conference on Parallel Processing*, pages 1–12. ACM, 2021.
[12] Michael Mathieu, Mikael Henaff, and Yann LeCun. Fast Training of Convolutional Networks through FFTs. In *Proceedings of the 2rd International Conference on Learning Representations (ICLR)*, 2014.
[13] Nicolas Vasilache, Jeff Johnson, Michael Mathieu, and et al. Fast Convolutional Nets with fbfft: A GPU Performance Evaluation. In *Proceedings of the 3rd International Conference on Learning Representations (ICLR)*, 2015.
[14] Zhiyi Zhang, Pengfei Zhang, Zhuopin Xu, Bingjie Yan, and Qi Wang. Im2col-Winograd: An Efficient and Flexible Fused-Winograd Convolution for NHWC Format on GPUs. In *Proceedings of the 53rd International Conference on Parallel Processing*, page 1072–1081, 2024.
[15] Liancheng Jia, Yun Liang, Xiuhong Li, Liqiang Lu, and Shengen Yan. Enabling efficient fast convolution algorithms on gpus via megakernels. *IEEE Transactions on Computers*, 69(7):986–997, 2020.
[16] Juan Yepez and Seok-Bum Ko. Stride 2 1-D, 2-D, and 3-D Winograd for Convolutional Neural Networks. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 28(4):853–863, 2020.
[17] Da Yan, Wei Wang, and Xiaowen Chu. 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)*, page 109–123, 2018.
[18] Yun Liang, Liqiang Lu, Qingcheng Xiao, and Shengen Yan. Evaluating Fast Algorithms for Convolutional Neural Networks on FPGAs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(4):857–870, 2020.
[19] S. Kala, Babita R. Jose, Jimson Mathew, and S. Nalesh. High-performance cnn accelerator on fpga using unified winograd-gemm architecture. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27(12):2816–2828, 2019.
[20] Chengcheng Huang, Xiaoxiao Dong, Zhao Li, Tengteng Song, Zhenguo Liu, and Lele Dong. Efficient Stride 2 Winograd Convolution Method Using Unified Transformation Matrices on FPGA. In *Proceedings of 2021 International Conference on Field-Programmable Technology (ICFPT)*, pages 1–9, 2021.
[21] Jung-Woo Chang, Saehyun Ahn, Keon-Woo Kang, and Suk-Ju Kang. Towards Design Methodology of Efficient Fast Algorithms for Accelerating Generative Adversarial Networks on FPGAs. In *2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 283–288, 2020.
[22] Chen Yang, Yizhou Wang, Xiaoli Wang, and Li Geng. WRA: A 2.2-to-6.3 TOPS Highly Unified Dynamically Reconfigurable Accelerator Using a Novel Winograd Decomposition Algorithm for Convolutional Neural Networks. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 66(9):3480–3493, 2019.
[23] Jinming Lu, Hui Wang, Jun Lin, and Zhongfeng Wang. WinTA: An Efficient Reconfigurable CNN Training Accelerator With Decomposition Winograd. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 71(2):634–645, 2024.
[24] Jingbo Jiang, Xizi Chen, and Chi-Ying Tsui. Accelerating large kernel convolutions with nested winograd transformation. In *2023 IFIP/IEEE 31st International Conference on Very Large Scale Integration (VLSI-SoC)*, pages 1–6, 2023.
[25] Di Huang, Xishan Zhang, Rui Zhang, Tian Zhi, and et al. DWM: A decomposable Winograd method for convolution acceleration. In *AAAI Conference on Artificial Intelligence*, volume 34, page 4174–4181, 2020.
[26] Shmuel Winograd. *Arithmetic complexity of computations*, volume 33. 1980.
[27] Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *Proceedings of the 3rd International Conference on Learning Representations (ICLR)*, 2015.
[28] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778. IEEE, 2016.
[29] Jie Hu, Li Shen, Samuel Albanie, Gang Sun, and Enhua Wu. Squeeze-and-Excitation Networks. In *Proceedings of IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 7132–7141, 2018.
[30] Zhuang Liu, Hanzi Mao, Chao-Yuan Wu, Christoph Feichtenhofer, Trevor Darrell, and Saining Xie. A convnet for the 2020s. In *2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 11966–11976, 2022.
[31] Jia Deng, Wei Dong, Richard Socher, and et al. Imagenet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 248–255, 2009.
[32] Alex Krizhevsky. Cifar10. http://www.cs.toronto.edu/ kriz/cifar.html.
[33] Monica D. Lam, Edward E. Rothberg, , and Michael E. Wolf. 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)*, Santa Clara, CA, USA, 1991. ACM.
[34] Alex Krizhevsky. cuda-convnet2, 2014. https://code.google.com/archive/p/cuda-convnet2/.
[35] Yun Liang, Liqiang Lu, Qingcheng Xiao, and Shengen Yan. Evaluating fast algorithms for convolutional neural networks on fpgas. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(4):857–870, 2020.
[36] Bizhao Shi, Jiaxi Zhang, Zhuolun He, and et al. Efficient Super-Resolution System With Block-Wise Hybridization and Quantized Winograd on FPGA. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 42(11):3910–3924, 2023.
[37] Xiaohan Ding, Xiangyu Zhang, Jungong Han, and Guiguang Ding. Scaling up your kernels to 31x31: Revisiting large kernel design in cnns. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 11963–11975, June 2022.
[38] Xiaohan Ding, Yiyuan Zhang, Yixiao Ge, and et al. UniRepLKNet: A Universal Perception Large-Kernel ConvNet for Audio, Video, Point Cloud, Time-Series and Image Recognition. In *2024 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 5513–5524, 2024.