AirWino: Optimized Winograd Convolution for Accelerating CNN Inference on ARMv8 Processors

Haoyuan Gui[1,2], Xiaoyu Zhang[1,2], Yifan Zhang[1,2], Ximeng Fu[1,2], Shiqi Sun[1,2], Leisheng Li[2,3], Huiyuan Li[2,3]*

[1] University of the Chinese Academy of Sciences
[2] Institute of Software, Chinese Academy of Sciences
[3] Key Laboratory of System Software, Institute of Software, Chinese Academy of Sciences
huiyuan@iscas.ac.cn

Abstract

As Convolutional Neural Networks (CNNs) continue to gain traction in deep learning, Winograd convolution has emerged as a key algorithm to enhance computational efficiency. Although ARM-based CPUs are increasingly prevalent in mobile devices, embedded systems and HPC servers, existing 2D Winograd convolution implementations for ARM often leave room for improvement in transformation efficiency, computational throughput, and overall versatility. Furthermore, the lack of tailored 3D Winograd convolution implementations for ARM architectures stems from the additional complexity of supporting higher-dimensional kernels. AirWino introduces a set of novel optimizations covering transformations, data layouts, micro-kernel computations, and parallelization strategies for both 2D and 3D Winograd convolution. It supports FP32 and FP16 precisions with filter sizes of 3 and 5, targeting a broad range of applications. Evaluations on four distinct ARM platforms show that AirWino consistently outperforms state-of-the-art libraries across various experimental scenarios and hardware configurations, highlighting its efficiency and portability.

Code — https://github.com/guihaoyuan/AirWino

Introduction

Convolutional Neural Networks (CNNs) have achieved remarkable success in computer vision, medical image segmentation, video analysis, and beyond (Simonyan and Zisserman 2014; Quan, Hildebrand, and Jeong 2021; Krizhevsky, Sutskever, and Hinton 2012; Szegedy et al. 2015; Tran et al. 2015; Çiçek et al. 2016; Milletari, Navab, and Ahmadi 2016), including both 2D and 3D convolution tasks. However, their convolutional layers are computationally expensive and often dominate the overall execution time of modern deep networks. This issue is particularly pronounced for 3D CNNs, where the data size and model complexity increase significantly. To tackle this bottleneck, various approaches have been proposed to accelerate convolutions. One prominent method is the Winograd minimal filtering algorithm (Winograd 1980), originally developed for signal processing and later adopted by Lavin and Gray (2016) for CNNs. By substantially reducing the number of multiplications required for small-kernel convolutions, Winograd convolution has become a popular choice for both training and inference, and has been integrated into many mainstream libraries. Concurrently, a variety of optimizations have been explored to further enhance Winograd's performance on different architectures (Jia et al. 2018; Dolz et al. 2023; Yan, Wang, and Chu 2020; Liu, Yang, and Lai 2021; Wei et al. 2020; Budden et al. 2017; Meng et al. 2022; Li et al. 2021a; Wang et al. 2024).

Despite these advancements, existing Winograd-based solutions designed for modern CPUs (Research 2016; Tencent 2024; ARM 2024; Intel 2024; Jia et al. 2018; Dolz et al. 2023; Dukhan 2024; Budden et al. 2017; Meng et al. 2022; Li et al. 2021a; Wang et al. 2024) either only support 2D Winograd convolution or fail to provide dedicated optimizations for 3D convolutions. Their tuning effort chiefly targets matching ISA features, leaving the fine-grained dataflow across Winograd stages largely unexploited.

Meanwhile, ARM architectures—widely used in mobile computing and embedded systems—are increasingly being adopted in high-performance computing (HPC); in certain CNN inference tasks, these processors can offer lower latency and improved cost-effectiveness compared to GPUs (Li, Paolieri, and Golubchik 2024; Aghapour et al. 2024; Chheda et al. 2023; Tabuchi et al. 2021). However, current Winograd implementations on ARM (Tencent 2024; ARM 2024; Meng et al. 2022; Dukhan 2024; Intel 2024) demonstrate suboptimal parallel and computational efficiency, with limited support for higher-dimensional convolutions, varied filter sizes, and mixed floating-point precisions. Notably, mainstream ARM-compatible libraries, such as ncnn (Tencent 2024) and the ARM Compute Library (ACL) (ARM 2024) only offer naive solutions for 3D convolutions. Simultaneously, OneDNN (Intel 2024) resorts to im2col-based general matrix multiplications (GEMM) method for 3D tasks on ARM, lacking further architecture-specific optimizations. These limitations significantly impede CNN inference performance on ARM-based platforms.

To address these challenges, we introduce AirWino, a highly optimized, scalable Winograd convolution implementation specifically targeting ARMv8 architectures. AirWino provides comprehensive supporting for Winograd convolution and optimized assembly micro-kernels for

*Corresponding author
Copyright © 2026, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

all three Winograd stages—input & filter transformation, batched GEMM, and output transformation—covering multiple Winograd variants. To minimize dependency stalls between load and compute instructions, AirWino integrates a double-buffering (ping-pong) strategy (Wei et al. 2022; Wang et al. 2015; Jiang et al. 2017; Wei et al. 2024) within its batched GEMM micro-kernel, paired with a customized data layout.

We evaluate AirWino's performance on four distinct platforms —Huawei Kunpeng 920 (HiSilicon 2019), AWS Graviton2 (Services 2025), Phytium 2000+ (Su et al. 2018), and Raspberry Pi 5—comparing it against six state-of-the-art convolution libraries (ARM 2024; Tencent 2024; Dukhan 2024; Intel 2024; Meng et al. 2022; Wang et al. 2023). Experimental results show that AirWino consistently outperforms these libraries across a variety of scenarios and hardware configurations.

The main contributions of this paper are summarized as follows:

- A comprehensive, high-performance implementation of Winograd convolution is proposed, encompassing both 2D and 3D operations, supporting FP32 and FP16 precisions, and accommodating filter sizes of 3 and 5, with seamless integration into mainstream deep learning frameworks.
- A novel Winograd-aware, dimension-specific optimization framework for Winograd convolution is presented, delivering superior performance compared to state-of-the-art convolution libraries.
- A rigorous performance analysis model is established to systematically guide the selection of optimal parameters and execution modes (Fused vs. Non-Fused).

Background

Convolution is a discrete linear transformation that extracts local features by sliding a filter over the input feature map and computing dot products. While 2D convolution moves the filter only across height $\times$ width, 3D convolution moves it across height, width, and depth, capturing spatiotemporal or volumetric patterns at the cost of higher memory and compute. In 3D CNNs the input tensor has shape $[N, C, D, H, W]$, the filter tensor $[K, C, R_D, R_H, R_W]$, and the output is $[N, K, D', H', W']$.

The Winograd minimal filter algorithm (Winograd 1980) accelerates convolution by algebraically reducing the required multiplications. In Winograd's notation, $F(m, r)$ denotes a 1D convolution that produces m output values with an r-tap filter, implying an input length of $m+r-1$. For such an $F(m, r)$, the algorithm lowers the multiplication count by a factor of $\frac{m \times r}{m+r-1}$. The filter vector g and input vector d are first transformed to the Winograd domain via matrices G and B; after element-wise multiplication, the result is mapped back to the spatial domain with matrix A:

$$o = [(gG^T) \odot (dB)]A, \quad (1)$$

where $\odot$ symbolizes element-wise multiplication, and matrices B, G and A are obtained once per (m,r) by applying the Chinese Remainder Theorem to the convolution polynomials, yielding fixed transform coefficients. Winograd convolution can be extended to $\mathbb{N}$-dimensions by independently transforming each dimension n of the filter and input tensor using transformation matrices G_n, B_n and A_n through tensor-matrix mode-n multiplication ($\times_n$, defined in (Kolda and Bader 2009)):

$$o = [(g \times_{n=1}^{\mathbb{N}} G_n^T) \odot (d \times_{n=1}^{\mathbb{N}} B_n)] \times_{n=1}^{\mathbb{N}} A_n. \quad (2)$$

For brevity, set $l_i = m_i + r_i - 1$ and $L = \prod_{i=1}^{\mathbb{N}} l_i$. In practical CNNs, inputs are tiled with the overlap-add (OLA) scheme so that every tile holds L elements, with neighbouring tiles sharing $r_i - 1$ elements along dimension i. After the Winograd transforms, every tile yields L Winograd-domain coefficients indexed by x. For any index x, accumulation over the C is expressed as the GEMM. Consequently, the b-th batch result of Winograd domain $\hat{O}$ can be reconstructed as:

$$\hat{O}_{k,t}^{(x)} = \sum_{c=1}^{C} U_{k,c}^{(x)} V_{c,t}^{(x)}, \quad (3)$$

where k indexes the output channel, c the input channel, and t the tile. Here $U_{k,c}^{(x)}$ is the x-th coefficient of the filter tensor after the $\mathbb{N}$-D transform $\times_{n=1}^{\mathbb{N}} G_n^T$, while $V_{c,t}^{(x)}$ is the corresponding coefficient of the b-th input tile t obtained via $\times_{n=1}^{\mathbb{N}} B_n$.

Related Work and Motivation

Building on Lavin and Gray's (2016) seminal adaptation of Winograd's algorithm to CNNs, subsequent work has pursued Winograd-based convolution optimizations along multiple directions. On GPUs, innovations span thread-level scheduling, Tensor-Core utilization, kernel fusion, and SASS-level tuning (Yan, Wang, and Chu 2020; Jia et al. 2020; Wei et al. 2020; Liu, Yang, and Lai 2021). On CPUs, TensorGEMM (Meng et al. 2022) and TEWMM (Li et al. 2021a) reduce transformation overheads by easing strided accesses, albeit at some cost to floating-point arithmetic intensity (FAI). Li et al. (2021a) further introduced NUMA-aware parallelism, while LoWino leverages the VNNI ISA for low-precision Winograd kernels on x86 (Li et al. 2021b; Wang et al. 2024). Jia et al. (2018) generalized the algorithm to $\mathbb{N}$-dimensional via a recursive formulation.

Despite these advances, most existing approaches still focus on aligning data layouts with SIMD widths or ISA constraints, overlooking the fine-grained dataflow between Winograd stages and the distinct optimization requirements of higher-dimensional convolution. On ARM platforms, 3D support remains nascent: libraries such as ncnn, ACL, and oneDNN provide functional kernels yet deliver only modest throughput. Jiang et al. (2023) proposed a hierarchical partitioning scheme for 3D convolution and a distributed inference solution, but it lacks operator-level optimizations. These omissions leave appreciable headroom that AirWino exploits through a Winograd-aware framework augmented with dimension-specific optimizations that address both 2D and 3D Winograd convolution workloads on ARM architectures.

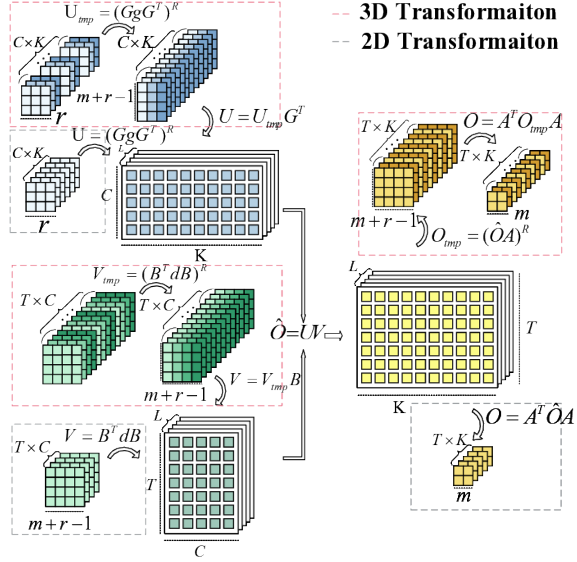

Figure 1: Overview of AirWino framework.

Design Architecture of AirWino

In this section, we provide a comprehensive overview of the design architecture of AirWino. Winograd convolution comprises three fundamental stages: Input & Filter Transformation, Computation, and Output Transformation. An overview of the AirWino framework is shown in Figure 1, where the symbol R denotes a $90°$ rotation of the tensor about its second spatial axis during the transformation of the third dimension. The total number of tiles, denoted by T, can be expressed as follows for the 2D and 3D cases, respectively:

$$\textbf{2D:}\quad T = \frac{(H - r_1 + 1)(W - r_2 + 1)}{m_1 m_2},$$
$$\textbf{3D:}\quad T = \frac{(H - r_1 + 1)(W - r_2 + 1)(D - r_3 + 1)}{m_1 m_2 m_3}. \tag{4}$$

In this framework, the computation stage utilizes batched GEMM, with the batch count distinguishing 2D from 3D convolutions. In contrast to the recursive 1D Winograd approach (Jia et al. 2018), which sacrifices throughput for dimensional generality, AirWino deploys dimension-specific optimizations.

In the 2D case, data is directly transformed into the destination domain within a micro-kernel. For 3D convolutions, data is first stored into an intermediate layout (denoted with subscript *tmp* in Figure 1), optimized to minimize strided memory accesses, enabling consecutive loads in the subsequent micro-kernel, which finalizes the transformation into the target domain.

AirWino provides fully hand-tuned ARMv8-A A64 micro-kernels for all three Winograd phases, leveraging customized register blocking and fine-grained register control to maximize throughput. Building on established assembly optimizations such as instruction scheduling and software prefetching (Yang et al. 2021; Wei et al. 2024), we introduce a novel series of optimizations, detailed next.

Winograd-Friendly Data Layout

Beyond merely facilitating vectorized loads/stores, AirWino introduces a Winograd-aware data layout co-designed with the execution flow of every stage. The scheme improves spatial locality both within each stage and across adjacent stages, keeping the working set resident in cache during batched GEMM. Transformed tensors are partitioned into blocked matrices governed by the hyper-parameters T_{blk}, C_{blk}, and K_{blk}.

For transformed inputs and filters, AirWino adopts a distinctive z-shaped ordering that guarantees strictly consecutive memory accesses throughout batched GEMM. Table 1 summarizes these Winograd-friendly layouts: entries in bold are used exclusively in the Non-Fused mode, and markers ①,② indicate the two alternative transformation strategies. We define $T_n = T - (T\%T_{blk})$; (α, β) denotes the GEMM micro-kernel configuration. The parameter θ denotes the SIMD lane count and is architecture-specific: on ARMv8 (128-bit SIMD) $\theta = 4$ for FP32 and $\theta = 8$ for FP16.

Variable	Data Layout
I	$[N][C][H][W]$ or $[N][C][D][H][W]$
V_{tmp}	①$[C/C_{blk}][C_{blk}/\theta][T_{blk}][l_1][l_2][l_3][\theta]$
	②$[C/C_{blk}][C_{blk}/\theta][l_1][T_{blk}][l_2][l_3][\theta]$
V	$\boldsymbol{[T_n/T_{blk}]}[L][C/C_{blk}][T_{blk}/\alpha][C_{blk}/\theta][\alpha][\theta]$
F	$[K][C][R_H][R_W]$ or $[K][C][R_D][R_H][R_W]$
U_{tmp}	①$[l_2][K_{blk}/\theta][C_{blk}][l_1][r_3][\theta]$
	②$[l_1][K_{blk}/\theta][C_{blk}][l_2][r_3][\theta]$
U	$[K/K_{blk}][L][C/C_{blk}][K_{blk}/\beta][C_{blk}][\beta]$
$\hat{O}$	$\boldsymbol{[K/K_{blk}][T_n/T_{blk}]}[L][K_{blk}/\theta][T_{blk}][\theta]$
$\hat{O}_{tmp}$	①$[K_{blk}/\theta][T_{blk}][m_3][l_2][l_1][\theta]$
	② $[K_{blk}/\theta][T_{blk}][m_3][l_1][l_2][\theta]$
O	$[N][K][H'][W']$ or $[N][K][D'][H'][W']$

Table 1: Winograd-friendly data layout employed in the main loop.

Transformation Optimization

The Winograd transform stages consume a considerable share of the overall runtime and are inherently memory-bound. AirWino alleviates this bottleneck by minimizing load latency and promoting contiguous memory access across all transformation phases.

AirWino selects one of two transformation strategies according to (m, r) and the available SIMD register number: ①**Register-Resident Tile (RRT)**: When a complete tile and its transformed output fit simultaneously in the registers, the transform is performed tile-by-tile in a single pass. ②**Buffered Row-Column (BRC)**: When register capacity is insufficient, AirWino first applies the row-wise (right-multiplication) transform, stores the intermediates in a cache-aligned buffer, and then completes the column-wise stage.

In 2D, the transformed results are written directly into the z-shaped GEMM-friendly format; in 3D, the two strategies

produce slightly different layouts for V_{tmp} (Table 1), yet both ensure that every run of l_3 consecutive elements along the D-axis is contiguous. This organization avoids strided accesses in the subsequent depth transform—likewise executed with RRT or BRC at the micro-kernel level—thereby boosting the performance of 3D CNN workloads.

The transformation matrices A, B, G are fixed for each $F(m, r)$ variant and exhibit pronounced sparsity and anti-symmetry; AirWino exploits these traits to prune redundant arithmetic (Jia et al. 2018; Li et al. 2021a; Wang et al. 2024). In the input transform, adjacent tiles overlap by $r - 1$ elements per spatial axis. AirWino maps these shared elements to specific registers, so they are fetched once and reused once, cutting load traffic along the W-axis. Larger r values amplify the benefit as the overlap widens. The reuse policy matches the chosen transform strategy: Figure 2 illustrates the RRT workflow for $(m = 2, r = 3)$ and BRC workflow for $(m = 4, r = 3)$.

Batched GEMM Design

AirWino stores Winograd-domain tensors in a z-shaped layout that preserves contiguous memory access for GEMM operation, without any extra data packing. For each of the L matrix pairs, the kernel performs the block matrix multiplication $(T_{blk} \times C_{blk}) \cdot (C_{blk} \times K_{blk})$, yielding a $(T_{blk} \times K_{blk})$ block that is accumulated into the output. To sustain peak throughput, we design a Winograd-specific GEMM micro-kernel paired with a double-buffer streaming scheme, overlapping data movement with computation.

Micro-kernel design Each time the micro-kernel is invoked, an $\alpha \times C_{blk}$ slice of V is multiplied by a $C_{blk} \times \beta$ slice of U to yield an $\alpha \times \beta$ output tile. A double-buffer (ping-pong) scheduling is employed in the micro-kernel to overlap loading operations with fused multiply-add (FMA) instructions: while one register group feeds the FMAs, the other preloads the next operands. Each NEON register holds θ scalars, laid out as constant T with varying C for V and constant C with varying K for U. Pipeline saturation requires reserving α registers for the forthcoming V slice, $\frac{\beta}{\theta}$ for the next U slice, and $\frac{\alpha \times \beta}{\theta}$ for the accumulators. Given ARMv8's 32-register NEON file, the register budget therefore must satisfy

$$2 \times \alpha + \frac{2 \times \beta}{\theta} + \frac{\alpha \times \beta}{\theta} \leq 32. \tag{5}$$

Owing to the characteristic sizing of the K dimension in CNN workloads, and to eliminate edge cases along this axis, we constrain β such that $\beta \% 4 = 0$.

Across θ iterations, our approach requires α loads for V and $(\frac{\beta}{\theta} \times \theta)$ load for U. It executes $\alpha \times \beta$ scalar-vector FMAs (two floating-point operations each) and postpones write-back operation until the loop terminates. The resulting average FAI is therefore

$$\frac{2 \times \alpha \times \beta}{\alpha + \beta}. \tag{6}$$

To handle skinny-and-tall matrices, eliminate T dimension edge cases, and align parallel granularity with both matrix shape and thread count, AirWino offers a suite of GEMM micro-kernels (see Table 2) that are adaptively selected to optimize performance.

	$\theta = 4$ (FP32)			$\theta = 8$ (FP16)	
	Case A	Case B	Case C	Case D	Case E
Main	4×16	7×8	10×4	7×16	10×8
Edge	$\{1\text{~}3\} \times 16$	$\{1\text{~}6\} \times 8$	$\{1\text{~}9\} \times 4$	$\{1\text{~}6\} \times 16$	$\{1\text{~}9\} \times 8$

Table 2: Multiple GEMM micro-kernel variants implemented in AirWino.

Pipeline design Figure 4 depicts the ping-pong execution pipeline and its register blocking scheme for the FP32 micro-kernel configured as 4×16. Registers `v0-v7` load the input matrix, `v8-v15` load the filter matrix, and `v16-v31` accumulate results. Each operand pool is split into two equal sets; while one set feeds the FMAs, the other preloads the next operands, implementing a double-buffer schedule. With $\theta = 4$ lanes per register, a full iteration comprises eight stages (2θ): in stage #1 a 4×4 input block is loaded into `v0-v3` and the matching 4×4 filter block into `v8-v11`. Each input register then participates in θ consecutive scalar-vector FMAs, during which the alternate set completes its prefetch. The two filter sets swap roles every stage. Other FP32 configurations follow the same pipeline, differing only in register partitioning. The FP16 micro-kernel retains the pattern but spans 16 stages per iteration (2θ with $\theta = 8$).

Cache blocking analysis GEMM routines is a pivotal part in Winograd convolution, the choice of blocking parameters is critical to performance. Our analysis balances the cache hierarchy, loop ordering, and several objectives—chiefly maximizing the GEMM kernel's FAI. This paper details the Fused mode; the same principles apply equally to the Non-Fused variant.

In Figure 3 the yellow rectangle represents the entire batched GEMM. Let $\mathbb{C}$ be the cache capacity. During each pass over the C dimension the L2 cache must hold (i) one $T_{blk} \times C$ block of V; (ii) two $C_{blk} \times K_{blk}$ blocks of U (active and prefetched); and (iii) the corresponding $T_{blk} \times K_{blk}$ block of $\hat{O}$. Because ARM L2 is unified for instructions and data, these footprints must satisfied:

$$T_{blk} \times K_{blk} + T_{blk} \times C + 2 \times C_{blk} \times K_{blk} < \mathbb{C}_{L2}. \tag{7}$$

Within the GEMM kernel, AirWino multiplies $\frac{T_{blk}}{\alpha}$ slices of V (each sized $\alpha \times C_{blk}$) with a single $C_{blk} \times \beta$ slice of U. The resulting L1 constraint is

$$2 \times \alpha \times C_{blk} + \beta \times C_{blk} + T_{blk} \times K_{blk} \leq \mathbb{C}_{L1}. \tag{8}$$

Additionally, the following divisibility constraints must hold:

$$T_{blk} \% \alpha = 0, \quad C_{blk} \% (2 \times \theta) = 0, \quad K_{blk} \% \beta = 0. \tag{9}$$

Each kernel invocation loads $T_{blk} \times C$ block of V from last level cache (LLC), a $C \times K_{blk}$ block of U from main memory, and reads/writes the $T_{blk} \times K_{blk}$ block of $\hat{O}$; the resulting FAI of the GEMM kernel is:

$$\frac{2 \times T_{blk} \times C \times K_{blk}}{2 \times T_{blk} \times K_{blk} + T_{blk} \times C + C \times K_{blk}}. \tag{10}$$

AirWino employs a performance analysis model to determine optimal cache blocking parameters under the constraints in Eqs. (7)–(9), using Eq. (10) as one of several optimization criteria. Next section details the model.

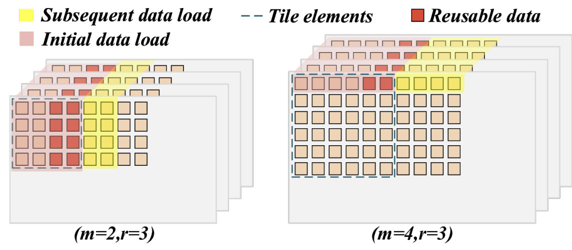

Figure 2: Schematic overview of the two distinct strategies for input transformation in AirWino.

Fused Mode	Non-Fused Mode

<pre>for bt=0 to T_n step T_blk
  for bc=0 to C step C_blk
    for ic=0 to C_blk step θ
      Input Transform d2
      Input Transform d1
  for bk=0 to K step K_blk
    for i=0 to L step 1
      for bc=0 to C step C_blk
        for u=0 to K_blk step β
  GEMM   for v=0 to T_blk step α
  Kernel   GEMM micro-kernel
  Output Transform d1
  Output Transform d2</pre>
	<pre>for bt=0 to T_n step T_blk
	  for bc=0 to C step C_blk
	    for ic=0 to C_blk step θ
	      Input Transform d2
	      Input Transform d1
	for bk=0 to K step K_blk
	  for i=0 to L step 1
	    for bt=0 to T_n step T_blk
	      for bc=0 to C step C_blk
	        GEMM Kernel
	for bk=0 to K step K_blk
	  for bt=0 to T_n step T_blk
	    Output Transform d1
	    Output Transform d2</pre>

Figure 3: Pseudocode of AirWino's Fused (left) and Non-Fused (right) implementations.

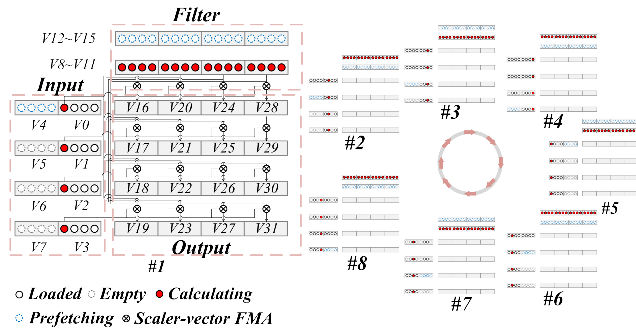

Figure 4: Register blocking and ping-pong pipeline procedure for the 4×16 (FP32) micro-kernel.

Parallel Strategies

AirWino employs a multi-dimensional OpenMP scheme that scales to any power-of-two thread count. Threads are mapped chiefly to the axis T and the channel axes C and K, with the dimension N activated only when additional parallelism is needed.

Fused mode. A static 2D thread grid is mapped to all three stages. Its first dimension enumerates the $\frac{T_n}{T_{blk}}$ subsets and is split across P_T thread groups of $\frac{P}{P_T}$ threads each. Within a group, the input transform is further partitioned over C into $\frac{C}{\theta}$ subtasks, whereas the GEMM and output stages partition over K into $\frac{K}{K_{blk}}$ subtasks. When T supplies ample work, threads parallelize only along T; if T is severely under-subscribed, the schedule falls back to parallelism on C and K, while intermediate cases preserve the full multi-dimensional parallelism.

Non-Fused mode. The three independent stages are parallelized separately. Collapsing the input-transform loop nest yields $\frac{T}{T_{blk}} \times \frac{C}{\theta}$ subtasks; batched GEMM exposes $\frac{K}{K_{blk}} \times L$ subtasks; and the output transform presents $\frac{T}{T_{blk}} \times \frac{K}{K_{blk}}$ subtasks.

Performance Analysis Model

We develop a recursive model that estimate AirWino's runtime under any parameter choice and selects the optimal configuration together with the execution mode (loop ordering). The model rests on two premises:

Assumption 1: The platform is equipped with a shared LLC; all threads at a given parallel level share the same cache slice.

Assumption 2: The overhead for loading the original input and transformed filter data from main memory remains constant under different parameter settings, and thus does not affect parameter selection. It is excluded from the performance analysis.

For a $d-$dimensional parallel framework, let the problem scale be $\Phi = \Phi\big(T, C, K, P\big)$, and the parameter set be $\Psi = \Psi\big(T_{blk}, C_{blk}, K_{blk}, P_T, \alpha, \beta, m, r\big)$. The estimated time overhead $\mathscr{T}$ is defined recursively as

$$\mathscr{T}^i(\Phi,\Psi) = \begin{cases} \mathbb{T}^i_{comp} + \mathbb{T}^i_{mem} + \mathbb{T}^i_{sync}, & \text{if } i = 1, \\ \mathbb{T}^i_{comp}(\mathscr{T}^{i-1}) + \mathbb{T}^i_{mem} + \mathbb{T}^i_{sync}, & \text{if } i \le d, \end{cases} \tag{11}$$

where $\mathbb{T}_{comp}$, $\mathbb{T}_{mem}$, and $\mathbb{T}_{sync}$ denote the time spent on computation, data transfer, and synchronization, respectively.

Fused Mode ($d = 2$). The same derivation applies to the non-fused case. Let $\mathbb{O}_{in}$ and $\mathbb{O}_{out}$ be the FLOP counts of the input and output transformations (determined by $F(m,r)$), and let $\mathscr{R}_{in}$, $\mathscr{R}_{gemm}$ and $\mathscr{R}_{out}$ be their attainable throughputs in GFLOP/s.

When $i = 1$, Fused Mode utilizes $\frac{P}{P_T}$ threads to carry out input transformation, GEMM, and output transformation in parallel. The sub-task sizes are $\mathbb{O}_{in}\big(T_{blk} \times \theta\big)$, $T_{blk} \times K_{blk} \times C \times L$ and $\mathbb{O}_{out}\big(T_{blk} \times K_{blk}\big)$, respectively. Temporary data produced in these stages is stored in the LLC, and we denote by $\mathbb{B}$ the bandwidth of the corresponding memory hierarchy. Because threads operating in this dimension are load-balanced, $\mathbb{T}^1_{sync}$ is negligible. Consequently, for $\mathbb{T}^1_{comp}$ and $\mathbb{T}^1_{mem}$, we have:

$$\begin{cases} \mathbb{T}^1_{comp} = \frac{\mathbb{O}_{in}(T_{blk}\times\theta)\times C\times P_T}{\mathscr{R}_{in}\times\theta\times P} + \frac{T_{blk}\times K_{blk}\times C\times L\times K\times P_T}{\mathscr{R}_{gemm}\times K_{blk}\times P} \\ \qquad + \frac{\mathbb{O}_{out}(T_{blk}\times K_{blk})\times K\times P_T}{\mathscr{R}_{out}\times K_{blk}\times P} \\ \mathbb{T}^1_{mem} = \frac{T_{blk}\times C\times L\times K}{\mathbb{B}_{LLC}\times K_{blk}} + \frac{T_{blk}\times K_{blk}\times L\times K}{\mathbb{B}_{LLC}\times K_{blk}}. \end{cases} \tag{12}$$

When $i = 2$, $\mathbb{T}^2_{comp} = \lfloor \frac{T_n}{T_{blk} \times P_T} \rfloor \times \mathscr{T}^1$ and $\mathbb{T}^2_{mem} = 0$. Since $\frac{T_n}{T_{blk}}$ may not be divisible by P_T, and T may not be divisible by T_{blk}, we define:

$$\begin{cases} \mathbb{T}^2_{sync} = \eta \times \mathscr{T}^1(\Psi_{rem}) + \mathscr{T}^1(\Phi_{edge}, \Psi_{edge}) \\ \Psi_{rem} = \Psi_{rem}\big(T_{blk}, C_{blk}, K_{blk}, \frac{P}{P_T}, \alpha, \beta, m, r\big) \\ \Psi_{edge} = \Psi_{edge}\big(T\%T_{blk}, C_{blk}, K_{blk}, \frac{P}{P_T}, \alpha, \beta, m, r\big) \\ \eta = 1 \ \textbf{if} \ \frac{T_n}{Tb}\%P_T \ \textbf{else} \ 0. \end{cases} \tag{13}$$

We exhaustively explore Ψ under the cache constraints (Eqs. 8-9) and restrict (α, β) to the micro-kernels in Table 2. For each candidate we evaluate $\mathscr{T}^2$ via Eq. 11 and keep the minimizer Ψ^{opt}. The specific values of $\mathscr{R}_{in}$, $\mathscr{R}_{gemm}$ and $\mathscr{R}_{out}$ are estimated from a Roofline model (Williams, Waterman, and Patterson 2009) using the kernel FAI at the given scale. Finally, the model chooses the faster loop ordering by comparing $\mathscr{T}_{\text{Fused}}$ with $\mathscr{T}_{\text{Non-Fused}}$.

Experiments

ID	C	K	H/W	R	ID	C	K	H/W	D	R
1	64	64	224	3	12	32	96	28		5
2	128	128	112	3	13	32	128	14		5
3	256	256	56	3	14	64	128	56	16	3
4	512	512	28	3	15	256	256	28	8	3
5	512	512	14	3	16	512	512	14	4	3
6	64	64	640	3	17	32	64	130	114	3
7	128	128	320	3	18	64	128	62	54	3
8	256	256	160	3	19	128	256	30	26	3
9	512	512	80	3	20	32	32	64	32	5
10	1024	1024	40	3	21	64	64	32	16	5
11	96	256	27	5	22	128	128	16	8	5

Table 3: Benchmarked convolutional layers.

Hardware configuration. Evaluations were performed on four ARM platforms—three HPC servers and one embedded board:

- **Kunpeng 920** 48 cores; 64 KB L1 I/D and 512 KB private L2 per core; 48 MB shared L3; 2.6 GHz.
- **AWS Graviton2 (c6g.16xlarge)** 64 cores; 64 KB L1 I/D and 1 MB private L2 per core; 32 MB shared L3; 2.5 GHz.
- **Phytium 2000+** 64 cores; 64 KB L1 I/D per core; 2 MB L2 shared by each four-core cluster; no L3; 2.2 GHz.
- **Raspberry Pi 5** 4 cores; 64 KB L1 I/D and 512 KB private L2 per core; 2 MB shared L3; 2.4 GHz.

Benchmarked layers. We selected representative layers from various mainstream convolutional neural networks—including VGG-16 (Simonyan and Zisserman 2014), FusionNet (Quan, Hildebrand, and Jeong 2021), AlexNet (Krizhevsky, Sutskever, and Hinton 2012), GoogLeNet (Szegedy et al. 2015), C3D (Tran et al. 2015), UNet 3D (Çiçek et al. 2016), and V-Net (Milletari, Navab, and Ahmadi 2016)—to assess AirWino's performance on both 2D and 3D convolutions with filter sizes 3 and 5. The detailed parameters for each layer appear in Table 3.

Baseline Libraries. We benchmark AirWino against six state-of-the-art ARM-optimized CNN libraries. Table 4 summarizes their support for Winograd configuration, numerical precision, 5×5 filters, and 3D convolutions. On ARM platforms, oneDNN delegates 2D convolutions to the ACL, while NDIRECT—though not Winograd-based—offers a highly tuned direct-convolution micro-kernel and is therefore included as a baseline.

Lib	Wino3 × 3	Prec	5 × 5	3D (Conv method)
ncnn	F2/F4/F6	32/16	im2col/direct	✓(naive)
ACL	F4	32/16	WinF2	✓(naive)
NNPACK	F6	32	FFT	✗
FastConv	F6	32	im2col	✗
OneDNN	ACL	32/16	ACL	✓(im2col)
NDIRECT	-	32	direct	✗
AirWino	F2/F4/F6	32/16	WinF2/F4	✓(Wino)

Table 4: Baseline libraries and AirWino capabilities. Notation: F2/F4/F6 denote the Winograd configurations $F(2^2, 3^2)$, $F(4^2, 3^2)$ and $F(6^2, 3^2)$, respectively. A precision tag of "32/16" indicates support for both FP32 and FP16, whereas "32" indicates FP32 only; WinF2/WinF4 correspond to $F(2^2, 5^2)$ and $F(4^2, 5^2)$ configurations for 5×5 filters.

Layer-wise Evaluation

We evaluated the layer-wise performance of AirWino against other libraries on both 2D and 3D benchmark layers ($R = 3$) under an FP32 configuration. Two execution regimes were tested: (i) $N = 1$ with 8 threads (4 on the Raspberry Pi 5); and (ii)N set to the total number of cores on each platform, using the same number of threads. To contrast Winograd with direct convolution we also include the NDIRECT, but only for regime (ii), as it does not support parallelization within the N-loop. The results in Figure 5, show that AirWino consistently outperforms the other libraries—most notably on 3D workloads— demonstrating the efficiency and portability of its Winograd-aware, dimension-specific design across both HPC servers and embedded devices.

To further showcase the versatility of AirWino in convolution inference, we performed two additional experiments on AWS Graviton2: (1) **Mixed precision**, $R = 3$. Inputs and outputs remain in FP32, while computation uses FP16. AirWino is compared with ncnn, ACL, and oneDNN; results appear in Figure 6. (2) **FP32**, $R = 5$. Figure 7 confirms that AirWino again delivers the best performance. Across all configurations AirWino maintains a clear lead, underscoring its strong, general-purpose acceleration of Winograd convolution.

End-to-end Evaluation

AirWino was integrated into PyTorch and evaluated on Kunpeng 920 ($N = 8$, 48 threads). We measured end-to-end latency for the VGG-16 and UNet-3D models, comparing AirWino with PyTorch back-ends that invoke ACL, oneDNN, im2col+OpenBLAS. Table 5 shows that AirWino halves the

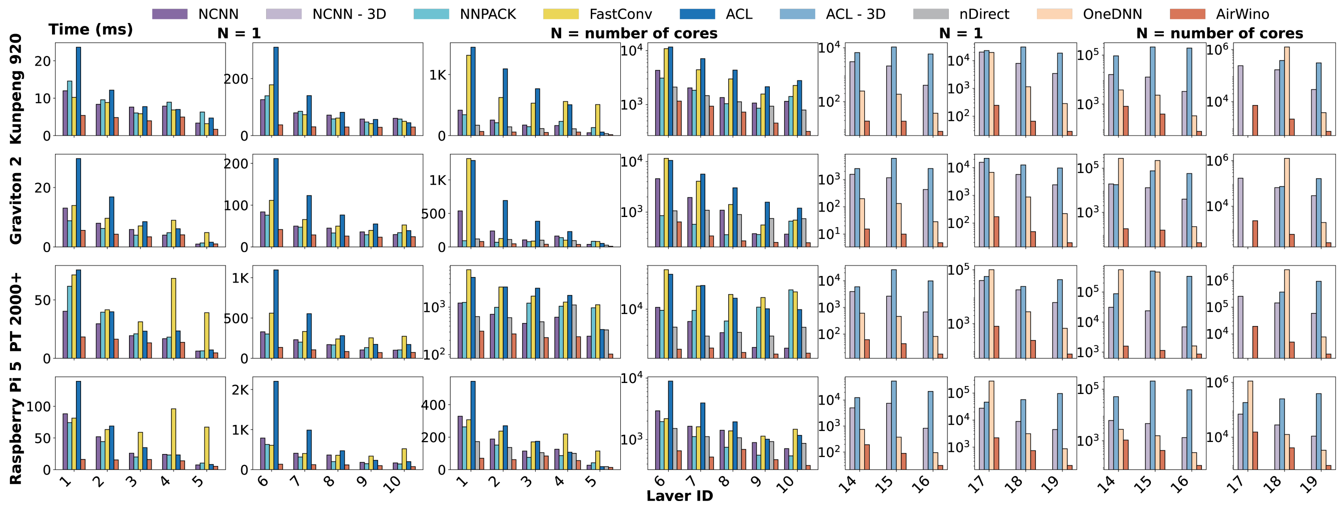

Figure 5: Layer-wise evaluation of AirWino vs. other libraries (FP32, $R = 3$): 2D (left four columns) and 3D (right four columns) layers on four distinct ARM platforms.

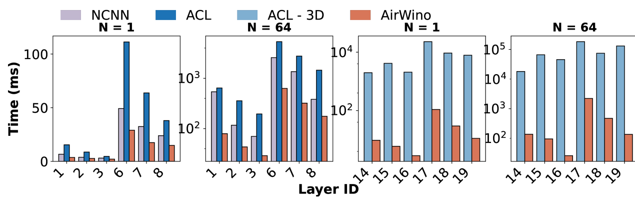

Figure 6: Layer-wise mixed-precision performance evaluation. ($R = 3$)

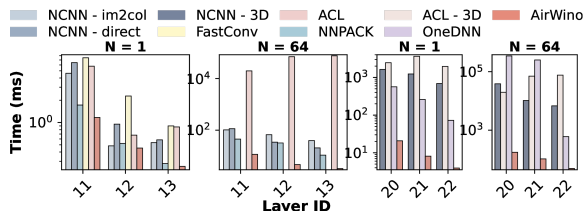

Figure 7: Layer-wise FP32 performance evaluation. ($R = 5$)

VGG-16	AirWino	ACL	im2col+OpenBLAS
Time (ms)	477.25	898.31	1389.42
UNet-3D	**AirWino**	**OneDNN**	**im2col+OpenBLAS**
Time (ms)	6778.59	122835.88	84129.55

Table 5: End-to-end inference latency (ms) on Kunpeng 920.

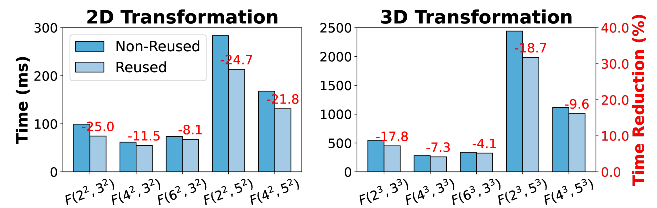

Figure 8: Input-transform reuse efficiency. Right axis: time reduction (%) vs. non-reuse baseline.

latency of VGG-16 and cuts UNet-3D by an order of magnitude, confirming its effectiveness in real-world inference workloads.

Data Reuse Gains in Input Transformation

To quantify the benefit of our reuse strategy, we measured the input-transform time on Kunpeng 920 for two representative, large-scale layers—layer 6 and layer 17 from Table 3. These layers incur relatively high transformation overhead, making them particularly sensitive benchmarks. Figure 8 compares AirWino (with reuse) against an otherwise identical baseline that reloads all data. The observed reductions track the theoretical ratio $\frac{r_1-1}{m_1+r_1-1}$. For 3D, the larger L introduces extra strided accesses and slightly reduces the gain. Even so, reuse cuts input-transform time by 8.1% to 25.0% in 2D and by 4.1% to 18.7% in 3D relative to the no-reuse baseline.

Conclusions

We present AirWino, a high-performance Winograd convolution implementation optimized for ARMv8 architectures. AirWino introduces a comprehensive suite of Winograd-aware optimizations and applies dimension-specific tuning for both 2D and 3D operators. Layer-wise benchmarks on four representative platforms, together with end-to-end evaluations via PyTorch integration, confirm its superior efficiency, portability, and practical value over state-of-the-art baselines. Although this study targets ARMv8, the proposed methodology is architecture-agnostic and readily applicable to other platforms.

Acknowledgments

The authors gratefully acknowledge the support of the National Natural Science Foundation of China (NSFC) under Grant No. 12471348 and the ISCAS Basic Research Project under Grant No. ISCAS-PYFX-202302.

References

Aghapour, E.; Sapra, D.; Pimentel, A.; and Pathania, A. 2024. ARM-CO-UP: ARM COoperative Utilization of Processors. *ACM Transactions on Design Automation of Electronic Systems*, 29(5): 1–30.

ARM. 2024. Compute Library. Accessed: 2024-12-17.

Budden, D.; Matveev, A.; Santurkar, S.; Chaudhuri, S. R.; and Shavit, N. 2017. Deep tensor convolution on multicores. In *International Conference on Machine Learning*, 615–624. PMLR.

Chheda, S.; Curtis, A.; Siegmann, E.; and Chapman, B. 2023. Performance study on CPU-based machine learning with PyTorch. In *Proceedings of the HPC Asia 2023 Workshops*, 24–34.

Çiçek, Ö.; Abdulkadir, A.; Lienkamp, S. S.; Brox, T.; and Ronneberger, O. 2016. 3D U-Net: Learning dense volumetric segmentation from sparse annotation. In *Medical Image Computing and Computer-Assisted Intervention–MICCAI 2016: 19th International Conference, Athens, Greece, October 17-21, 2016, Proceedings, Part II 19*, 424–432. Springer.

Dolz, M. F.; Martínez, H.; Castelló, A.; Alonso-Jordá, P.; and Quintana-Ortí, E. S. 2023. Efficient and portable Winograd convolutions for multi-core processors. *The Journal of Supercomputing*, 79(10): 10589–10610.

Dukhan, M. 2024. NNPACK. Accessed: 2024-12-28.

HiSilicon. 2019. Huawei Kunpeng 920.

Intel. 2024. oneDNN. Accessed: 2024-12-17.

Jia, L.; Liang, Y.; Li, X.; Lu, L.; and Yan, S. 2020. Enabling efficient fast convolution algorithms on GPUs via MegaKernels. *IEEE Transactions on Computers*, 69(7): 986–997.

Jia, Z.; Zlateski, A.; Durand, F.; and Li, K. 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*, 109–123.

Jiang, J.; Huang, Z.; Huang, D.; Du, J.; Chen, L.; Chen, Z.; and Lu, Y. 2023. Hierarchical model parallelism for optimizing inference on many-core processor via decoupled 3D-CNN structure. *ACM Transactions on Architecture and Code Optimization*, 20(3): 1–21.

Jiang, L.; Yang, C.; Ao, Y.; Yin, W.; Ma, W.; Sun, Q.; Liu, F.; Lin, R.; and Zhang, P. 2017. Towards highly efficient DGEMM on the emerging SW26010 many-core processor. In *2017 46th International Conference on Parallel Processing (ICPP)*, 422–431. IEEE.

Kolda, T. G.; and Bader, B. W. 2009. Tensor decompositions and applications. *SIAM Review*, 51(3): 455–500.

Krizhevsky, A.; Sutskever, I.; and Hinton, G. E. 2012. Imagenet classification with deep convolutional neural networks. *Advances in Neural Information Processing Systems*, 25.

Lavin, A.; and Gray, S. 2016. Fast algorithms for convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 4013–4021.

Li, D.; Huang, D.; Chen, Z.; and Lu, Y. 2021a. Optimizing massively parallel Winograd convolution on arm processor. In *Proceedings of the 50th International Conference on Parallel Processing*, 1–12.

Li, G.; Jia, Z.; Feng, X.; and Wang, Y. 2021b. Lowino: Towards efficient low-precision Winograd convolutions on modern cpus. In *Proceedings of the 50th International Conference on Parallel Processing*, 1–11.

Li, Z.; Paolieri, M.; and Golubchik, L. 2024. Inference latency prediction for CNNs on heterogeneous mobile devices and ML frameworks. *Performance Evaluation*, 165: 102429.

Liu, J.; Yang, D.; and Lai, J. 2021. Optimizing Winograd-based convolution with tensor cores. In *Proceedings of the 50th International Conference on Parallel Processing*, 1–10.

Meng, J.; Zhuang, C.; Chen, P.; Wahib, M.; Schmidt, B.; Wang, X.; Lan, H.; Wu, D.; Deng, M.; Wei, Y.; et al. 2022. Automatic generation of high-performance convolution kernels on ARM CPUs for deep learning. *IEEE Transactions on Parallel and Distributed Systems*, 33(11): 2885–2899.

Milletari, F.; Navab, N.; and Ahmadi, S.-A. 2016. V-net: Fully convolutional neural networks for volumetric medical image segmentation. In *2016 Fourth International Conference on 3D Vision (3DV)*, 565–571. Ieee.

Quan, T. M.; Hildebrand, D. G. C.; and Jeong, W.-K. 2021. Fusionnet: A deep fully residual convolutional neural network for image segmentation in connectomics. *Frontiers in Computer Science*, 3: 613981.

Research, C. 2016. FALCON Library: Fast Image Convolution in Neural Networks on Intel Architecture.

Services, A. W. 2025. Amazon EC2 Instances Powered by AWS Graviton Processors.

Simonyan, K.; and Zisserman, A. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*.

Su, X.; Liao, X.; Jiang, H.; Yang, C.; and Xue, J. 2018. SCP: Shared cache partitioning for high-performance GEMM. *ACM Transactions on Architecture and Code Optimization (TACO)*, 15(4): 1–21.

Szegedy, C.; Liu, W.; Jia, Y.; Sermanet, P.; Reed, S.; Anguelov, D.; Erhan, D.; Vanhoucke, V.; and Rabinovich, A. 2015. Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 1–9.

Tabuchi, A.; Shirahata, K.; Yamazaki, M.; Kasagi, A.; Honda, T.; Kurihara, K.; Kawakami, K.; Tabaru, T.; Fukumoto, N.; Kuroda, A.; et al. 2021. The 16,384-node parallelism of 3D-CNN training on an arm CPU based supercomputer. In *2021 IEEE 28th International Conference on*

High Performance Computing, Data, and Analytics (HiPC), 152–161. IEEE.

Tencent. 2024. ncnn. Accessed: 2024-12-17.

Tran, D.; Bourdev, L.; Fergus, R.; Torresani, L.; and Paluri, M. 2015. Learning spatiotemporal features with 3d convolutional networks. In *Proceedings of the IEEE International Conference on Computer Vision*, 4489–4497.

Wang, F.; Jiang, H.; Zuo, K.; Su, X.; Xue, J.; and Yang, C. 2015. Design and implementation of a highly efficient dgemm for 64-bit armv8 multi-core processors. In *2015 44th International Conference on Parallel Processing*, 200–209. IEEE.

Wang, P.; Yang, W.; Fang, J.; Dong, D.; Huang, C.; Zhang, P.; Tang, T.; and Wang, Z. 2023. Optimizing Direct Convolutions on ARM Multi-Cores. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 1–13.

Wang, X.; Li, G.; Jia, Z.; Feng, X.; and Wang, Y. 2024. Fast convolution meets low precision: Exploring efficient quantized Winograd convolution on modern CPUs. *ACM Transactions on Architecture and Code Optimization*, 21(1): 1–26.

Wei, C.; Jia, H.; Zhang, Y.; Xu, L.; and Qi, J. 2022. IATF: An input-aware tuning framework for compact BLAS based on ARMv8 CPUs. In *Proceedings of the 51st International Conference on Parallel Processing*, 1–11.

Wei, C.; Jia, H.; Zhang, Y.; Yao, J.; Li, C.; and Cao, W. 2024. IrGEMM: An Input-Aware Tuning Framework for Irregular GEMM on ARM and X86 CPUs. *IEEE Transactions on Parallel and Distributed Systems*.

Wei, H.; Liu, E.; Zhao, Y.; and Yu, H. 2020. Efficient non-fused Winograd on GPUs. In *Advances in Computer Graphics: 37th Computer Graphics International Conference, CGI 2020, Geneva, Switzerland, October 20–23, 2020, Proceedings 37*, 411–418. Springer.

Williams, S.; Waterman, A.; and Patterson, D. 2009. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4): 65–76.

Winograd, S. 1980. *Arithmetic complexity of computations*, volume 33. Siam.

Yan, D.; Wang, W.; and Chu, X. 2020. Optimizing batched Winograd convolution on GPUs. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 32–44.

Yang, W.; Fang, J.; Dong, D.; Su, X.; and Wang, Z. 2021. LIBSHALOM: Optimizing small and irregular-shaped matrix multiplications on ARMv8 multi-cores. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 1–14.