Exploration of Convolution Acceleration Algorithms: From Im2Col to Winograd

Performance Optimization Report

October 30, 2025

Abstract

This report documents an iterative journey to optimize convolution operations, a cornerstone of deep learning. The path involved continuously identifying performance bottlenecks and introducing targeted mathematical tools or optimization strategies to address them. The exploration moved from leveraging established techniques (GEMM via Im2Col) to mathematically complex approaches (FFT), and finally to a specialized arithmetic optimization (Winograd), which ultimately proved to be the performance champion due to its superior efficiency and scalability.

1 The First Stop: Leveraging GEMM – Im2Col

1.1 Initial Inspiration and Attempt

Inspired by industry practices, the initial approach focused on transforming convolution into a highly optimized General Matrix Multiplication (GEMM).

- **Implementation Idea:** The Im2Col method flattens all local receptive fields (patches) into a massive column matrix ($\mathbf{M}_{\text{col}}$). The convolution kernel is flattened into a row matrix, effectively turning the convolution into an efficient GEMM operation.
- **Initial Optimization:** To pursue maximum speed, I employed Numba JIT compilation with `parallel=True` enabled for automatic multi-threading.

1.2 First Bottleneck: The Large Image Performance Trap

Initial results were encouraging: performance on small images (32×32) showed an approximate 3x speedup compared to the baseline. This validated the theoretical efficiency of the GEMM approach.

However, when the dataset was switched to large resolution images (512×512), performance degraded sharply to only 0.8x the baseline speed.

Bottleneck Analysis:

- **Memory and Bandwidth Consumption:** The first step of Im2Col—building the gigantic $\mathbf{M}_{\text{col}}$ matrix—incurred prohibitively large time and memory bandwidth overhead on large images, completely neutralizing the subsequent GEMM acceleration.
- **Low Cache Hit Rate:** The complex, non-contiguous access patterns to the original image and the complex write pattern to the new $\mathbf{M}_{\text{col}}$ matrix resulted in a very low CPU cache line hit rate.
- **Limitation of Self-Implementation:** The self-implemented GEMM lacked fine-grained optimizations and could not compete with highly optimized library functions.

2 The Second Stop: Bypassing Data Rearrangement – FFT

2.1 Shift in Exploration Direction: Attacking Complexity

The failure of Im2Col indicated that extra data rearrangement and memory overhead were fatal flaws for large images. The focus shifted: Could the theoretical complexity of the operation be reduced using pure mathematics?

- **Introducing FFT:** The Fast Fourier Transform (FFT) can convert convolution in the spatial domain (complexity $O(N^2)$) into multiplication in the frequency domain ($O(N \log N)$). For large N, this offers a massive theoretical advantage.

2.2 New Bottleneck: The Scalability Trap

Regrettably, FFT showed poor multi-threaded scalability in practical testing. Even with an increase in threads, the acceleration effect was extremely limited.

Bottleneck Analysis (Amdahl's Law):

- **Core Steps are the Bottleneck:** In FFT-based convolution, the most computationally intensive steps are the FFT and IFFT transforms themselves, both having a complexity of $O(N \log N)$.
- **Limited Optimization Space:** These $O(N \log N)$ transformation steps are often already highly parallelized in underlying libraries. According to Amdahl's law, since the majority of time is spent on these already-parallelized core steps, parallelizing the remaining, less complex steps yields negligible overall speedup.

3 The Third Stop: Returning to the Source – Direct Convolution

3.1 Realization and Practice: Reducing Unnecessary Overhead

The FFT failure provided a critical insight: overly complex mathematical transformations, if not highly parallelizable or if they introduce hidden overheads, can become new bottlenecks.

- **Method:** I decided to return to the basics, focusing on minimizing the number and complexity of necessary operations, and purely parallelizing the core convolution loops.
- **Result:** Direct convolution demonstrated reasonably good performance, confirming that a simple, core, and efficiently parallelized approach was the correct direction.

4 The Fourth Stop: Multiplication Optimization – The Final Victory of Winograd

4.1 Seeking Higher Efficiency: Replacing Multiplication with Addition

Since direct convolution performed well, the next question was: Can the complexity of the core computation be further reduced while maintaining low overhead?

- **Introducing Winograd:** The fundamental idea of Winograd is to use number theory and polynomial concepts to significantly reduce the number of multiplications required for convolution, by replacing them with a larger number of additions. This is an elegant mathematical optimization that avoids the huge data rearrangement overhead of Im2Col.

- **Result:** The Winograd algorithm ultimately stood out, demonstrating the best performance and scalability. It emerged as the overall performance winner of the entire experiment.

4.2 Why Winograd is so Efficient: Detailed Analysis

The exceptional performance of the Winograd algorithm is a result of a synergy between a mathematically elegant core algorithm and aggressive compiler-level and micro-level optimizations.

4.2.1 Core Algorithm Optimisation: Winograd

This constitutes the primary optimisation, substantially reducing the number of multiplication operations in traditional convolution:

- **Reduction in Multiplication Count:** A conventional 3×3 convolution kernel requires $3 \times 3 = 9$ multiplications to compute a single output element. Winograd F(2×2,3×3) focuses on a 4×4 input tile and a 3×3 filter to produce a 2×2 output block. It reduces the multiplication count for the 4 output elements from $4 \times 9 = 36$ to just **$4 \times 4 = 16$** (in the transform domain). This provides a theoretical speedup factor of $36/16 = 2.25$.
- **Offline Weight Transformation:** The function $U = GWG^T$ pre-converts the convolution kernel W (the weights) into its transformed domain matrix U. This computationally intensive operation requires execution **only once** (e.g., at the start of inference) and can be reused for every input image, substantially accelerating the convolution computation in each iteration.

4.2.2 JIT Compilation Acceleration (Numba)

The software implementation achieves its high performance by leveraging the **Numba Just-In-Time (JIT) compiler** to turn high-level Python/NumPy code into efficient machine code.

- **@njit (No-Python Mode Compilation):** All core computational functions and all matrix multiplication auxiliary functions utilise '@njit'. This enables Python/NumPy code to run at **near C/C++ performance levels**.
- **prange and parallel=True (Parallel Processing):** Within the main function, parallel processing of image tiles is achieved by setting '@njit(parallel=True)' and using 'for tile_idx in prange(...)'. This enables different CPU cores to process distinct 4×4 blocks of the image simultaneously, resulting in **excellent multi-threaded scalability** across the input tiles.

4.2.3 Micro-optimisation and Memory Management

The code incorporates several minor optimisations tailored for the Numba environment to ensure generation of the fastest machine code:

- **Manual Expansion of Fixed-Size Matrix Multiplication:** Fixed-size matrix multiplication functions are defined. For the small, fixed-size matrices (like 4×4 or 2×4) used in Winograd filters, manually writing out the loops often yields more **efficient machine code** from Numba than using generic NumPy matrix multiplication operators, sometimes even enabling compiler optimizations like **automatic loop unrolling**.
- **fastmath=True:** Enabled in all matrix multiplication helper functions, allowing the compiler to perform more aggressive floating-point optimisations, typically sacrificing minimal precision for **higher speed**.

- **Memory Pre-allocation:** In main function, temporary matrices (e.g., ‘V_temp‘, ‘V‘, ‘X‘) are pre-allocated (using ‘np.zeros‘) **outside the loop and reused** within the inner loop. This avoids the overhead of frequent memory allocation and deallocation within the hottest inner loop.

5 Experimental Result Analysis and Key Performance Insights

This section details the performance of five convolution implementations: Naive Convolution, Im2Col (Self-Implemented GEMM), Im2Col (Library GEMM), FFT, and Winograd.

5.1 Correctness Validation

- **Test Method:** PyTorch’s standard `nn.Conv2d` function was used as the Ground Truth. All methods were tested under various conditions (input size, stride, and padding).
- **Conclusion:** The output results of all five methods were consistent with the standard PyTorch convolution result, validating the mathematical correctness and feasibility of the implemented algorithms.

5.2 Multithreaded Scalability Test

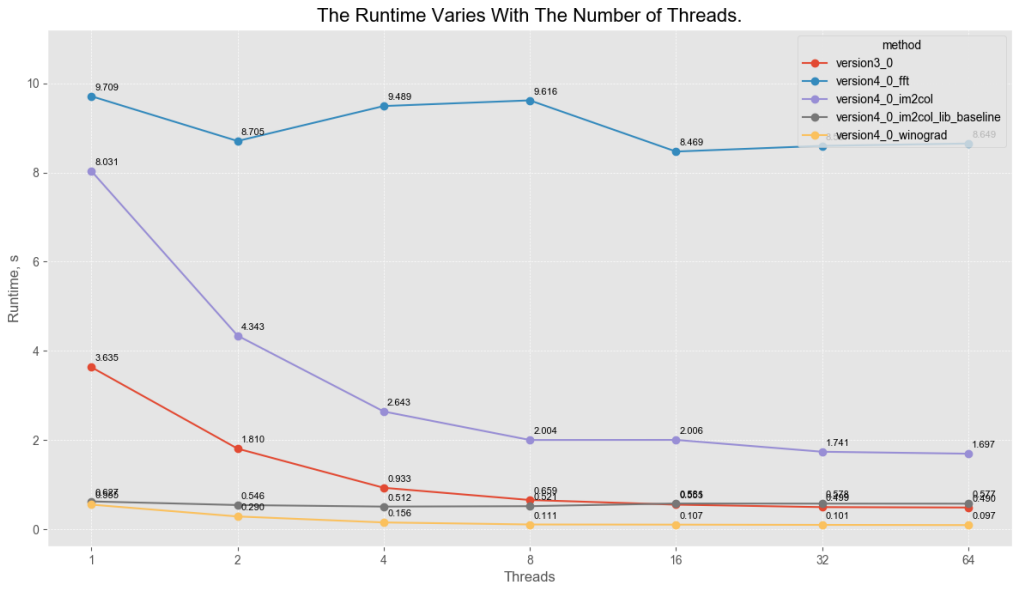

Figure 1: Visual representation of Multithreaded Scalability Test Results.(On Apple M4 Pro-12 Logic cores)

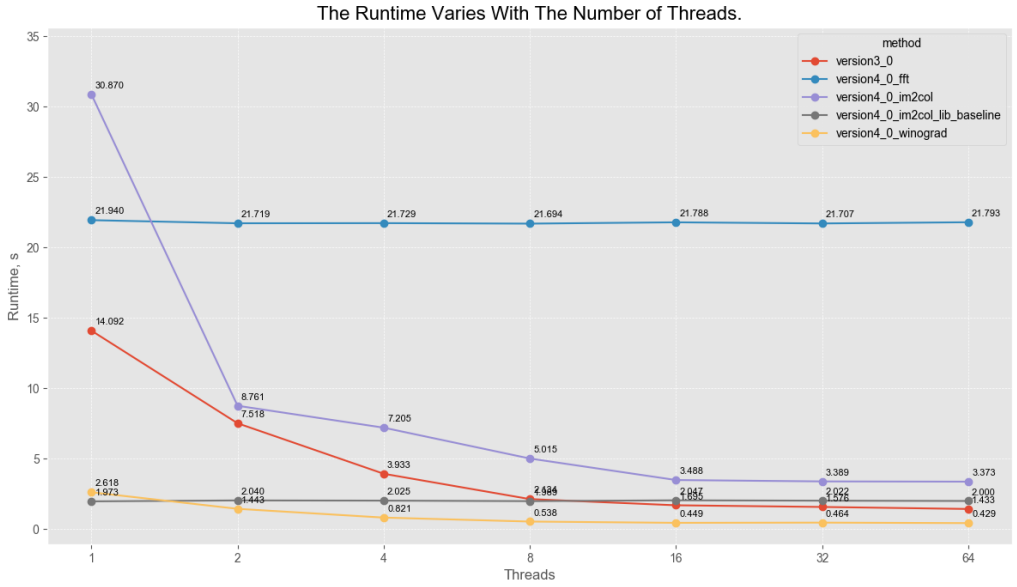

Figure 2: Visual representation of Multithreaded Scalability Test Results.(On Intel(R) Xeon(R) Gold 5120 CPU-28 Logic cores)

5.3 Scalability across Different Resolution Datasets

This test focused on performance trends across varying image sizes (small to large), revealing the algorithms' sensitivity to overhead.

Table 1: Scalability Across Different Resolution Datasets

Algorithm	Large-Size Performance Status	Causal Analysis
Winograd	Relatively Leading	Maintains a leading or near-leading position across all sizes, showing stable and excellent scalability.
Naive Conv.	Relative Improvement	Relative performance significantly improves with increasing image size, placing it second.Suggests its memory access pattern may be friendlier for large data.
Im2Col	Performance Collapse	Throughput sharply declines on large images. Key bottleneck: massive memory copying and huge memory footprint, with overhead completely offsetting GEMM's advantage.
FFT	Performance Collapse	Throughput sharply declines on large images. Key bottleneck: time overhead of the large FFT/IFFT transformations and complex memory access, leading to poor large-image performance.

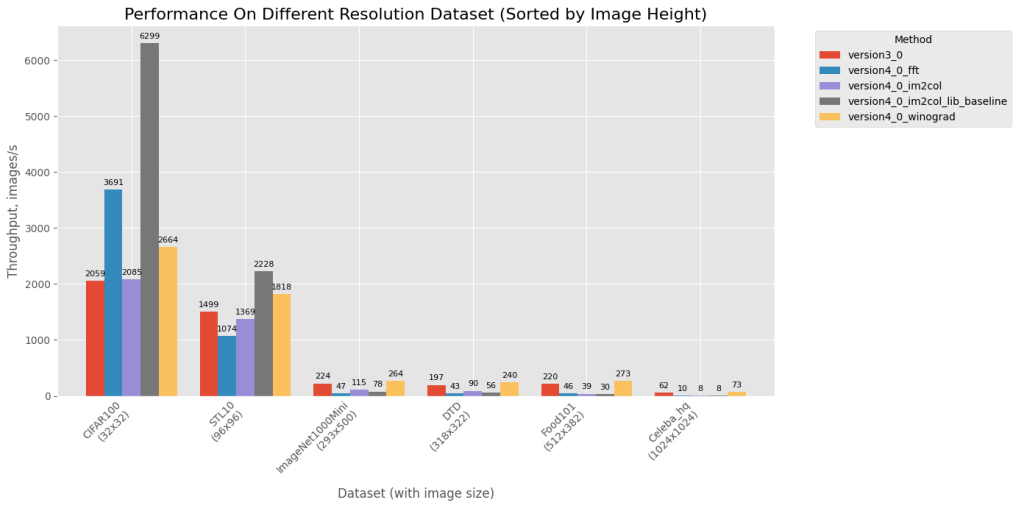

Figure 3: Visual representation of Scalability Across Different Resolution Datasets

5.4 Throughput under Common Specifications (32×32 and 96×96)

This test simulates the practical running performance in common small-to-medium specifications often found in deep learning.

Table 2: Throughput Under Common Specifications

Algorithm Version	32×32 Throughput	96×96 Throughput	Performance Analysis
Im2Col (Library)	Fastest	Leading/Near Leading	The fastest version in this experiment, fully demonstrating the huge potential of utilizing a highly optimized GEMM library.
Winograd	Excellent	Excellent	Provides consistent performance gain and is the most balanced fast algorithm for 32×32 and 96×96 sizes.
FFT	Excellent	Failed, Slower than Naive	Excellent performance on 32×32 images, but fails on the slightly larger 96×96, showing extreme sensitivity to image size.
Im2Col (Self-Implemented)	Worst	Worst	Confirms that the memory overhead and data rearrangement introduced by the convolution transformation must be compensated for by an extremely optimized GEMM to be worthwhile.

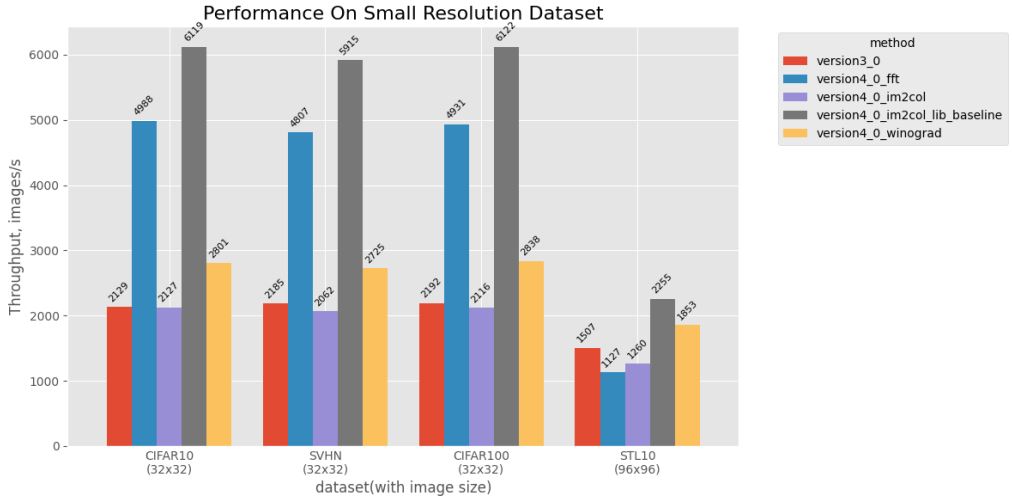

Figure 4: Visual representation of Throughput Under Common Specifications

6 Comprehensive Performance Summary

Table 3: Overall Algorithm Performance Summary

Algorithm	Performance Rank	Core Advantage	Core Disadvantage/Limitation
Winograd	Optimal	Stable, highly efficient, excellent scalability	Relatively complex implementation
Im2Col (Library)	Second Best	Extreme utilization of library GEMM parallelism and optimization	large-size performance collapse
Naive Conv.	Moderate	Memory friendly, relatively good large-size scalability	High theoretical complexity ($O(N^2)$)
FFT	Worst	Theoretically optimal complexity ($O(N \log N)$)	Highly size-sensitive, poor scalability
Im2Col (Self-Implemented)	Worst	Validates the Im2Col concept	Unoptimized GEMM, overhead cannot be compensated for

A environment info

A.1 environment info of ecetesla0

- OS: Linux 6.14.0-33-generic
- OS Version: 33 24.04.1-Ubuntu SMP PREEMPT DYNAMIC Fri Sep 19 17:02:30 UTC 2
- Architecture: x86
- Python Version: 3.12.3 (GCC 13.3.0)
- CPU: Intel(R) Xeon(R) Gold 5120 CPU @ 2.20GHz
- Physical Cores: 14
- Logical Cores (Threads): 28
- Max Frequency: 3200.00 MHz
- Total RAM: 92.97 GB

A.2 environment info of macbook pro

- OS: Darwin 25.0.0
- OS Version: Darwin Kernel Version 25.0.0: Wed Sep 17 21:41:26 PDT 2025; root:xnu-12377.1.9 141/RELEASE ARM64 T6041
- Architecture: arm64
- Python Version: 3.9.6 (Clang 17.0.0 (clang-1700.3.19.1))
- CPU: Apple M4 Pro
- Physical Cores: 12
- Logical Cores (Threads): 12
- Max Frequency: 4.00 MHz
- Total RAM: 48.00 GB