Exploring CPU and GPU Implementations of Winograd Convolution Algorithms for EdgeAI Inference

Jaydeep Gupta
RPTU Kaiserslautern
Dept. of Commercial Vehicle Technology
Kaiserslautern, Germany
niw15cud@rptu.de

***Abstract*—This work investigates the implementation of Winograd convolution algorithms on both CPU and GPU platforms in the context of inference. Specifically, the study presents a comparative analysis across multiple performance parameters, including latency, throughput, and power consumption. By focusing on these key performance metrics, the work underscores the suitability of Winograd-based convolution for deployment on resource-constrained embedded devices.**

***Index Terms*—Winograd Convolution, EdgeAI, CPU, GPU, Inference Optimization, Embedded AI**

I. Introduction

Convolutional Neural Networks (CNNs) have demonstrated superior performance compared to other neural network architectures in computer vision tasks. However, the increasing size and complexity of CNN models make them computationally intensive and time-consuming to execute on conventional hardware such as CPUs and GPUs. For time sensitive applications such as autonomous driving, surveillance drones, and robotics achieving low latency inference is critical [2]. In such scenarios, any delay in processing can compromise safety, reliability, and real-time decision-making.

To address this challenge, algorithmic optimizations such as the Winograd convolution algorithm have been proposed. Originally introduced by Andrew Lavin and Scott Gray, the Winograd algorithm significantly reduces the number of arithmetic operations required for small convolution kernels, especially the widely used 3×3 filters [1]. Research has shown that Winograd convolution can accelerate CNN computations by a factor of 2.25 to 3 compared to standard direct convolution [1]. This reduction in computational overhead directly translates into faster execution, thereby improving the overall latency of CNN inference pipelines—an essential requirement for modern real-time applications.

In this seminar, I investigated different frameworks capable of implementing Winograd convolution on selected hardware platforms, such as CPUs and GPUs. This analysis provides insights into the practical performance benefits and limitations of Winograd convolution across diverse computational environments, highlighting its potential to optimize CNN inference latency.

II. Background

Winograd convolution uses a minimum size of filter and reduces the number of multiplication operations, leading to reduction in expensive multiplication operation [1]. For instance, Convolution of $O = I * F$ where O is 2 x 2 output, I is 4 x 4 input, F is 3 x 3 kernel [1]. Total number of multiplications in this convolution will be 32, while same with Winograd convolution will be 16 (element-wise) multiplication through the following formula[1] .

$$O = A^T \left[\left(GFG^T \right) \odot \left(B^T I B \right) \right] A \tag{1}$$

where $\odot$ denotes element-wise multiplication, and A^T, G, and B^T are transformation matrices. However, Winograd convolution increases the number of addition which requires only one arithmetic operation, unlike in multiplication requires 2 arithmetic operations [3].

III. Implementation on CPU and GPU

A. CPU Implementation

In this work, I investigated the implementation of the Winograd convolution algorithm on an Intel CPU using two different frameworks, namely OneDNN and TVM. While OneDNN provides optimized primitives for various deep learning operations, the Winograd convolution was not supported for the specific CPU architecture under study. Consequently, OneDNN could not be used to evaluate Winograd-based execution in this context. To address this limitation, I employed the TVM framework, specifying LLVM as the compilation backend to generate optimized low-level code for the Intel CPU. Within TVM, two convolutional algorithms were implemented and systematically benchmarked: (i) the direct convolution as a baseline, and (ii) the TVM Winograd convolution explicitly defined using $topi.nn.conv2d\_winograd\_nchw$. The comparative evaluation across these approaches enabled a structured analysis of performance trade-offs, highlighting the effectiveness of the Winograd transformation when supported and properly scheduled on general-purpose CPUs.

B. GPU Implementation

To implement convolution on a GPU, I used the Winograd algorithm in cuDNN for expicit execution, the forward pass is configured with the dedicated function flag $CUDNN\_CONVOLUTION\_FWD\_ALGO\_WINOGRAD$. The workflow begins with creating and configuring the required cuDNN descriptors for input tensors, filters, and the convolution operation, including parameters such as stride, padding, and dilation. Once the convolution descriptor is set, the Winograd algorithm is explicitly selected as the forward algorithm. Before execution, it is necessary to query the required workspace size with $cudnnGetConvolutionForwardWorkspaceSize$, as Winograd often demands additional memory for intermediate transformations. The workspace is then allocated on the GPU, after which the convolution is executed through cudnnConvolutionForward, passing in the input data, filter weights, and allocated workspace. After computation, the workspace and descriptors must be released. It is important to mention that the advantages of using the Winograd convolution algorithm are most for small kernel sizes, particularly the commonly used 3×3 filters. However, its applicability is constrained by the GPU architecture and the cuDNN version being used, since not all platforms provide support for the Winograd algorithm. As a result, implementations must be designed with portability in mind, ensuring that alternative algorithms such as GEMM or FFT-based convolution can serve as fallbacks in cases where Winograd is not supported.

The experimental procedure involved repeatedly executing each convolution on the GPU. Performance metrics were systematically recorded, encompassing latency (ms), throughput (images/s) and power consumption (W). To enable a comprehensive comparison, both direct and Winograd convolution were implemented using custom `CUDA` kernels specifically optimized for GPU execution.

TABLE I
MODEL SETUP FOR EXPERIMENTS

Parameter	Value
Batch Size	64
Input Feature Map	$3 \times 32 \times 32$
Output Feature Map	$64 \times 32 \times 32$
Kernel Size	3×3
Stride, Padding	1

IV. RESULTS

The experimental evaluation was carried out on a workstation equipped with an Intel Core i5 9th Generation CPU and an NVIDIA GeForce GTX 1650 GPU, with the CIFAR-100 dataset serving as the input workload. To assess and compare the efficiency of different convolutional implementations, a set of key performance metrics was defined, including latency, throughput, and power consumption. In particular, the performance of direct convolution and Winograd convolution was systematically benchmarked on both the CPU and GPU.

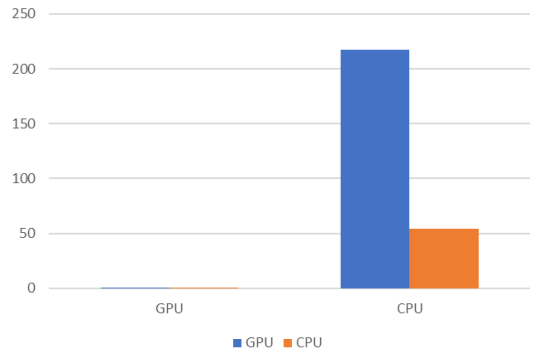

Fig. 1. Latency(ms) comparison between CPU and GPU implementations for direct and Winograd convolution.

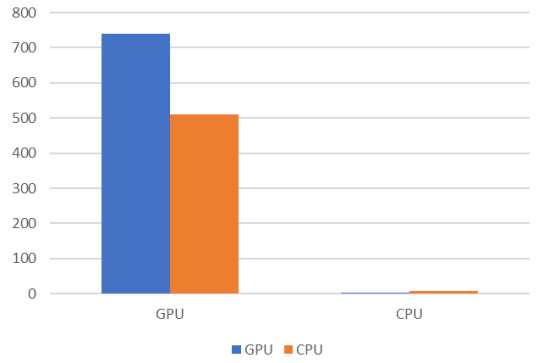

Fig. 2. Throughput (GFLOPS) comparison between CPU and GPU implementations.

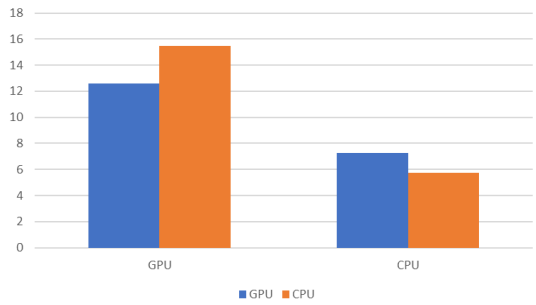

Fig. 3. Power consumption (watts) comparison between CPU and GPU implementations.

These findings highlight the trade-offs between CPU and GPU deployments in terms of performance versus energy constraints.

REFERENCES

[1] D. Yan, W. Wang, and X. Chu, "Optimizing batched Winograd convolution on GPUs," in *Proc. 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 32–44, 2020.

[2] Y. Shi, K. Yang, T. Jiang, J. Zhang, and K. B. Letaief, "Communication-efficient edge AI: Algorithms and systems," *IEEE Communications Surveys & Tutorials*, vol. 22, no. 4, pp. 2167–2191, 2020.

[3] X. Liu, J. Pool, S. Han, and W. J. Dally, "Efficient sparse-winograd convolutional neural networks," *arXiv preprint arXiv:1802.06367*, 2018.