End-to-End Deployment of Winograd-Based DNNs on Edge GPU

(Article begins on next page)

22 December 2024

Article

End-to-End Deployment of Winograd-Based DNNs on Edge GPU [†]

Pierpaolo Mori [1,2,*] , Mohammad Shanur Rahman [1], Lukas Frickenstein [1], Shambhavi Balamuthu Sampath [1], Moritz Thoma [1], Nael Fasfous [1], Manoj Rohit Vemparala [1], Alexander Frickenstein [1], Walter Stechele [3] and Claudio Passerone [2,*]

[1] BMW AG, 80809 Munich, Germany
[2] Department of Electronics and Telecommunications, Politecnico di Torino, 10129 Turin, Italy
[3] Chair of Integrated Systems, Technical University of Munich, 80333 Munich, Germany
[*] Correspondence: pierpaolo.mori@polito.it (P.M.); claudio.passerone@polito.it (C.P.)
[†] This paper is an extension of our paper published at the Winter Conference on Applications of Computer Vision (WACV) 2024 with the title "Wino Vidi Vici: Conquering Numerical Instability of 8-Bit Winograd Convolution for Accurate Inference Acceleration on Edge".

Abstract: The Winograd algorithm reduces the computational complexity of convolutional neural networks (CNNs) by minimizing the number of multiplications required for convolutions, making it particularly suitable for resource-constrained edge devices. Concurrently, most edge hardware accelerators utilize 8-bit integer arithmetic to enhance energy efficiency and reduce inference latency, requiring the quantization of CNNs before deployment. Combining Winograd-based convolution with quantization offers the potential for both performance acceleration and reduced energy consumption. However, prior research has identified significant challenges in this combination, particularly due to numerical instability and substantial accuracy degradation caused by the transformations required in the Winograd domain, making the two techniques incompatible on edge hardware. In this work, we describe our latest training scheme, which addresses these challenges, enabling the successful integration of Winograd-accelerated convolution with low-precision quantization while maintaining high task-related accuracy. Our approach mitigates the numerical instability typically introduced during the transformation, ensuring compatibility between the two techniques. Additionally, we extend our work by presenting a custom-optimized CUDA implementation of quantized Winograd convolution for NVIDIA edge GPUs. This implementation takes full advantage of the proposed training scheme, achieving both high computational efficiency and accuracy, making it a compelling solution for edge-based AI applications. Our training approach enables significant MAC reduction with minimal impact on prediction quality. Furthermore, our hardware results demonstrate up to a 3.4× latency reduction for specific layers, and a 1.44× overall reduction in latency for the entire DeepLabV3 model, compared to the standard implementation.

Keywords: CNN; Winograd convolution; hardware accelerator; quantization; NVIDIA; GPU

1. Introduction

Convolutional neural networks (CNNs) are widely used in computer vision tasks, achieving state-of-the-art performance in a wide range of applications, such as image classification [1], object detection [2], and semantic segmentation [3]. Their superior prediction quality made them the main choice for many real-world applications, from autonomous vehicles to facial recognition and medical imaging. However, as CNN architectures have evolved, their increasing complexity has led to an higher computational and memory demands. Edge hardware, which often comes with strict limitations in terms of power consumption, memory, and computational capacity, struggles to accommodate large-scale CNNs. The key challenge arises from the vast number of multiply–accumulate (MAC)

operations required by such models. Additionally, the memory required to store model parameters, including activations and weights, further exacerbates the problem, particularly on devices with limited memory resources. To address these challenges, model compression techniques, such as quantization and pruning, have become essential for enabling efficient inference on edge devices. CNN models are often overparameterized and present redundant weights that can be pruned, without affecting the prediction quality. Quantization has gained significant traction in both industry and academia as a practical solution for addressing the challenges associated with deploying convolutional neural networks (CNNs) on edge devices [4]. By reducing the bit-width of weights and activations, quantization allows CNNs to leverage low-precision arithmetic, significantly improving efficiency on hardware platforms with limited computational and memory resources.

There are two primary methods for applying quantization to CNNs: Post-Training Quantization (PTQ) and Quantization-Aware Training (QAT). In PTQ, a pre-trained model in full-precision (usually 32-bit floating-point) is converted to a lower bit-width representation, typically after a brief calibration phase involving a small set of training data. PTQ is a fast and straightforward approach, making it suitable for scenarios where access to the original training data is limited or where retraining is not feasible. On the other hand, Quantization-Aware Training (QAT) offers a more sophisticated solution by incorporating the quantization process directly into the training phase. QAT simulates the effects of quantization during training, allowing the model to learn how to adapt to the reduced precision. This process results in a quantized CNN that experiences minimal degradation in task-specific metrics, even at very low bit-widths (e.g., INT4 or INT8) [5,6].

With the ever-rising demand for fast inference, custom accelerators designed for specific layer types (e.g., pooling [7]) and optimized to operate on efficient fixed-point integers have garnered attention. Low-bit integer arithmetic is desirable as it is less complex and results in faster computation than floating point arithmetic. Despite the benefits brought by lower-bit quantization, when the hardware architecture used for the inference does not take advantage of lower-bit data representation (e.g., 8-bit hardware and 4-bit quantization), reduced bit width does not bring any further performance improvement (NVIDIA Jatson Nano [8], Snapdragon 8 Gen 1 [9], Google Coral [10]).

For general-purpose hardware that does not support quantization below 8-bit, reducing the latency of convolutional neural networks (CNNs) requires alternative methods, such as employing faster algorithms like the Winograd-based convolution. In a standard convolution, a filter moves across the input activation map, performing a series of multiply-accumulate (MAC) operations to generate a single output pixel at each position. In contrast, the Winograd-based convolution algorithm [11] significantly improves efficiency by computing multiple output pixels simultaneously. By leveraging mathematical transformations, Winograd-based convolution reduces the number of required MAC operations, thereby speeding up CNN inference on edge devices [12]. As an example, in the $F(4,3)$ variant of the Winograd algorithm, a $4\times$ reduction in the number of MAC operations can be achieved, compared to standard convolution. Winograd convolution consists of three stages: (1) input and weight transformation, (2) element-wise matrix multiplication (EWMM) of the transformed matrices, and (3) inverse transformation to produce the spatial output feature maps.

In the floating-point domain, the Winograd transformations introduce minimal numerical error, which does not significantly affect the model's prediction accuracy [13,14]. As a result, the floating-point Winograd algorithm has been widely adopted in the research community and optimized for various scenarios. For example, researchers have extended the algorithm to support additional layer types [15], introduced reuse schemes for greater efficiency [16,17], and explored pruning techniques to further reduce computational complexity [18,19]. However, when the Winograd algorithm is adopted in quantized domain, the numerical instability is exacerbated and causes severe task metric degradation [20]. In particular, when quantizing all the three stages of the Winograd algorithm described above, the limited bit width of all the operands cannot properly represent the wide ranges

in the Winograd domain, leading to overflow. For example, a 16-bit standard convolution ResNet-18 achieving 92% accuracy on CIFAR-10 would degrade down to 19.25% when using 16-bit Winograd $F(4, 3)$ convolution [20].

Several research efforts have attempted to address these challenges and enable accurate inference with quantized Winograd convolution. However, the solutions typically involve selective quantization, where only certain stages of the algorithm are quantized, while others remain at higher precision. While this approach helps preserve model accuracy, it introduces additional computational and hardware overhead.

In this work, we summarize and extend our research [21] on efficient and accurate quantized inference of CNNs adopting the Winograd algorithm. We tackle the challenges of quantized Winograd with the following contributions:

- We model the numerical error of quantized Winograd at training time, making the model aware of quantization errors and overflows in the Winograd domain.
- We introduced a trainable clipping factor for quantizing transformed parameters in the Winograd domain, resulting in a MAC operation reduction of 2.45× for ResNet-18-ImageNet with *only* ∼1 p.p. accuracy degradation.
- We designed an optimized 8-bit CUDA kernel for the F(4×4, 3×3) variant of the Winograd algorithms on an edge GPU. We took advantage of the efficient Tensor Cores to further speed-up the quantized algorithm, resulting in up to 3.41× latency reduction compared to the standard convolutional algorithm.

2. Related Works

2.1. Post-Training Winograd-Based Quantized CNNs

Li et al. [22] resorted to the Winograd $F(4, 3)$ algorithm to speed up inference time on CPUs. Full-precision transformations to/from the Winograd domain are used to minimize the accuracy degradation. Post-training quantization is then used to discretize the transformed weights and activations to 8-bit, performing only EWMM on 8-bit multipliers. The input and weights for all the convolutional layers are stored in 32-bit floating-point, demanding expensive computations for transformations and high memory bandwidth between two layers. Chikin et al. [23] propose to balance the data ranges of inputs and filters by scaling them channel-wise with balancing coefficients in order to equalize channel ranges to improve the quality of quantization. Furthermore, they also apply scaling factors per pixel within the tile to better map the transformed inputs/weights from floating-point to the quantized range. The channel balancing and channel-wise tile scaling factors increase the computational complexity of the transformation. Additionally, the pixel-wise scaling factors lead to the introduction of an expensive dequantization step before performing inverse transformation. Only the element-wise multiplication is quantized, similar to Li et al. [22], leading to huge hardware overhead.

Differently, this work implements CNNs with full 8-bit Winograd-based convolutions and only needs *one scaling factor* for the transformations of one layer.

2.2. Winograd-Aware Training (WAT)

Marques et al. [20] first included the Winograd algorithm in the training loop, making the model aware of the numerical instability problem. The authors made the Winograd transformation matrices trainable to reduce the accuracy degradation caused by quantization overflow. They further proposed wiNAS to search for Winograd-aware quantized networks by deciding the optimal Winograd tile size for each convolutional layer. However, as transformation matrices are trainable and maintained in floating-point representation, the computational cost in edge deployment gets expensive. Different Winograd tiles in various layers demand additional hardware logic to implement required transformations. Barabasz [24] resorts to Legendre polynomials to tackle numerical error for 8-bit Winograd quantization. In this approach, additional floating-point matrix multiplications are added along with standard Winograd transforms, increasing the complexity of the method. Moreover, the best accuracy values are achieved by making trainable Winograd transformation

matrices, increasing the complexity during CNN inference. Andri et al. [25] propose the use of pixel-wise scaling factors per tile, similar to those presented in [23], as a means to prevent quantization overflow in the Winograd-based convolutions. These scaling factors are efficiently determined during training through the use of powers-of-two values. Furthermore, they explore the different quantization levels at various stages of transformations and implement an 8-bit WAT pipeline using knowledge distillation. However, the quantized model still requires pixel-wise scaling factors, demanding additional dequantization to perform inverse transformation.

This work also proposes a novel WAT to achieve 8-bit quantized CNNs with no overhead in terms of hardware logic/storage, using only one scaling factor for all the transformations in one layer.

2.3. Winograd Algorithm on GPU

Castro et al. [26] propose an open-source implementation of the $F(2,3)$ Winograd algorithm on a GPU. The authors optimized the data layout in shared memory to improve the efficiency of the memory access and to reduce the thread divergence. The computation of each step is performed in single precision. However, the advantages of the Winograd $F(2,3)$ algorithm are limited compared to the larger $F(4,3)$. Liu et al. [27] presented an optimized Winograd kernel to accelerate the $F(6,3)$ algorithm on GPU, exploiting Tensor Cores. Each step of the Winograd algorithm (input transformation, weight transformation, EWMM, inverse transformation) is optimized to minimize the latency and enable the acceleration on Tensor Cores. Moreover, mixed precision (FP16) computation is adopted to minimize the numerical error introduced by the Winograd algorithm. However, the authors focused on server-grade GPUs and do not tackle 8-bit quantization.

Differently, this work focuses on the $F(4,3)$ Winograd algorithm, providing an optimized design to speed up the convolution on edge GPUs by exploiting Tensor Cores and 8-bit quantization.

3. Materials and Methods

This section presents the methodology for enhancing Winograd-based quantization on edge hardware, focusing on two core innovations. After a brief overview of quantized convolution and the Winograd convolution algorithm (Sections 3.1 and 3.2), Section 3.3 introduces the application of adaptive clipping factors in the Winograd domain. These clipping factors are applied to both weights and activations after transformation, effectively constraining the data range, reducing quantization error, and mitigating sources of numerical instability. Then, in Section 3.4, the design of a custom Winograd kernel optimized for a widely used edge GPU is presented, demonstrating the practical benefits of this approach in real-world edge applications. Together, these contributions showcase the feasibility and effectiveness of deploying quantized Winograd algorithms on edge devices.

3.1. Quantized Convolutional Algorithm

Consider a convolutional layer $l \in [1, \dots, L]$ in an L-layer CNN. The weights of the layer, $\mathcal{W}^l \in \mathbb{R}^{k_y \times k_x \times C_i \times C_o}$, are applied to an input feature map $\mathcal{A}^{l-1} \in \mathbb{R}^{H_i \times W_i \times C_i}$, where H_i, W_i and C_i are the spatial and channel dimension of the input. The kernel size is defined by k_y and k_x, and C_o is the number of output channels. The convolution between $\mathcal{W}^l$ and $\mathcal{A}^{l-1}$ generates the output feature map $\mathcal{A}^l \in \mathbb{R}^{H_o \times W_o \times C_o}$, with H_o, W_o, and C_o representing the output spatial and channel dimensions. The number of multiply and accumulate (MAC) operations for each layer can be calculated as as:

$$\#\text{MAC} = H_o \times W_o \times C_o \times k_x \times k_y \times C_i \tag{1}$$

Modern CNNs typically contain a vast number of parameters, which are commonly represented as 32-bit floating-point values during the training phase. While floating-point representation enables high precision, it poses significant challenges for inference on resource-constrained hardware due to its computational demands and limited memory

bandwidth. To mitigate these issues, a common approach is to quantize the weights and activations of the network to lower bit-width representations, such as 8, 4, 2, or even 1 bit. This process enhances throughput, decreases memory usage, and significantly reduces power consumption, making it more feasible to deploy complex CNN models in real-time applications. The quantization of a floating-point value x_f is described in Equations (2) and (3). For activation quantization, x_f is clipped between $[-c, +c]$, where c represents the trainable clipping threshold value for each layer and it is determined by the task-specific loss function of the CNN model [5]. Based on the determined c, a scaling factor SF_a is computed as $SF_a = c/(2^{N-1} - 1)$, where N represents the quantization bit width. When using the ReLU activation function, the range is clipped to $[0, c]$, with the scaling factor adjusted to $SF_a = c/(2^N - 1)$, representing the quantized value as an unsigned number.

$$x_{int} = \text{Q}_\text{C}(x_f) = \text{Round}(\text{Clip}(x_f, -c, +c)/SF_a) \tag{2}$$

Floating-point weights are quantized according to Equation (3), where the distribution is constrained to the range $[-1, +1]$ [6].

$$x_{int} = \text{Q}(x_f) = \text{Round}(x_f/SF_w) \tag{3}$$

Here, the scaling factor for the weights is given by $SF_w = 1/(2^{N-1} - 1)$. In order to deal with the discreteness of Equations (2) and (3) at training time, a straight-through estimator (STE) is used to update full-precision weights during backpropagation, while quantized values are used in the forward pass [28]. This paper focuses on 8-bit quantization, where all operands are represented as 8-bit integers.

3.2. Winograd Algorithm

The Winograd algorithm reduces the number of multiply–accumulate (MAC) operations required for convolution, offering significant computational reduction for CNNs [11]. The 2D Winograd algorithms are denoted as $F(m_x \times m_y, k_x \times k_y)$, where the output tile size is $m_x \times m_y$ and the kernel size is $k_x \times k_y$. Winograd convolution consists of three stages (Figure 1): first (1), the input activation and weight tiles are transformed to the Winograd domain. Then (2), the element-wise matrix multiplication (EWMM) is performed on the transformed tiles. Finally (3), the inverse transformation is applied to the resulting tiles in the Winograd domain, converting them back to the spatial domain to produce the output feature maps.

The standard convolutional algorithm requires $(m_x \times k_x) \times (m_y \times k_y)$ multiplications to produce an $m_x \times m_y$ output tile. In contrast, the 2D Winograd convolution requires only $(m_x + k_x - 1) \times (m_y + k_y - 1)$ multiplications. The $m_x \times m_y$ output tile is obtained through the algorithm flow shown in Figure 1 and Equation (4).

$$a^l = A^T[(Gw^lG^T) \odot (B^Ta^{l-1}B)]A \tag{4}$$

Here, a^{l-1}, w^l, and a^l represent the 2D tiles of the input feature map $\mathcal{A}^{l-1}$, of the weight tensor $\mathcal{W}^l$, and of the convolution output $\mathcal{A}^l$, respectively. The symbol $\odot$ represents the EWMM operator. B, G, A are the *constant* Winograd matrices responsible for transforming a^{l-1}, w^l and a^l to and from the Winograd domain [11]; the Winograd Matrices for the $F(4 \times 4, 3 \times 3)$ Winograd algorithm are reported in Equation (5).

$$B^T = \begin{bmatrix} 4 & 0 & -5 & 0 & 1 & 0 \\ 0 & -4 & -4 & 1 & 1 & 0 \\ 0 & 4 & -4 & -1 & 1 & 0 \\ 0 & -2 & -1 & 2 & 1 & 0 \\ 0 & 2 & -1 & -2 & 1 & 0 \\ 0 & 4 & 0 & -5 & 0 & 1 \end{bmatrix} \quad G = \begin{bmatrix} \frac{1}{4} & 0 & 0 \\ -\frac{1}{6} & -\frac{1}{6} & -\frac{1}{6} \\ -\frac{1}{6} & \frac{1}{6} & -\frac{1}{6} \\ \frac{1}{24} & \frac{1}{12} & \frac{1}{6} \\ \frac{1}{24} & -\frac{1}{12} & \frac{1}{6} \\ 0 & 0 & 1 \end{bmatrix}$$

$$A^T = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & 2 & -2 & 0 \\ 0 & 1 & 1 & 4 & 4 & 0 \\ 0 & 1 & -1 & 8 & -8 & 1 \end{bmatrix} \quad (5)$$

The transformed input tile $B^T a^{l-1} B$ and weights $G w^l G^T$ are of dimensions $r_x \times r_y$, where $r_x = m_x + k_x - 1$ and $r_y = m_y + k_y - 1$. For an entire convolutional layer, the number of MAC operations required by the Winograd algorithm is expressed in Equation (6).

$$\#\text{MAC}_{wino} = \left\lceil \frac{H_o}{m_y} \right\rceil \times r_y \times \left\lceil \frac{W_o}{m_x} \right\rceil \times r_x \times C_o \times C_i \quad (6)$$

Therefore, MAC reductions achieved with the Winograd algorithm for a convolutional layer can be computed as the ratio of Equations (1)–(6).
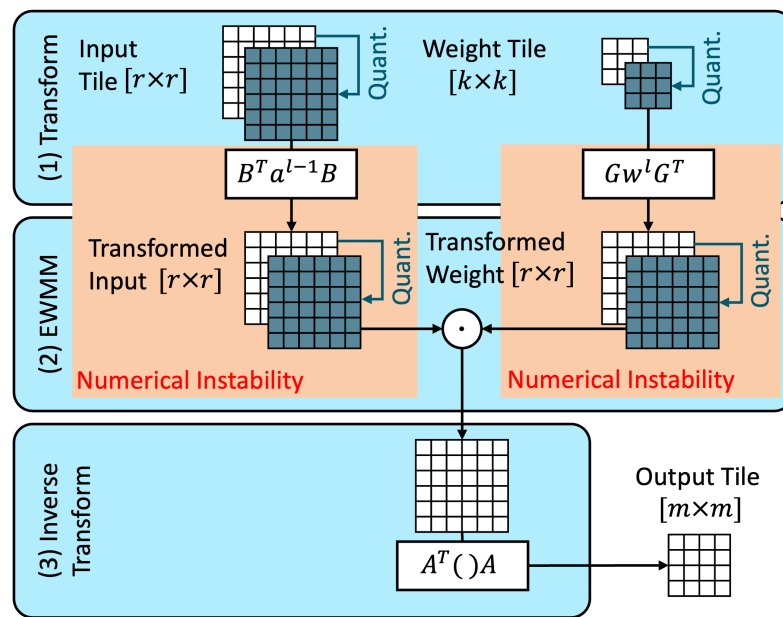

Figure 1. The three steps of the $F(4,3)$ Winograd algorithm: (1) input and weight transformation, (2) element-wise matrix multiplication (EWMM) of the transformed matrices, and (3) inverse transformation to produce the spatial output feature maps. The numerical instability due to quantization is highlighted.

While the Winograd algorithm can easily replace standard convolution in floating-point representations with minimal accuracy loss, reducing the data precision introduces numerical instability. This is especially evident with lower bit widths, where Winograd transformations cause significant numerical errors that degrade prediction quality. For instance, using a 16-bit integer variant of the Winograd algorithm $F(4 \times 4, 3 \times 3)$ leads to a 75 percentage point accuracy drop for ResNet-18 on CIFAR-10 [20]. This numerical error is caused by two main reasons: (1) numerical range *enlargement* after Winograd transform and (2) *rounding error* exacerbated by underutilizing the quantized range, as visualized in Figure 2a. The non-integer coefficients in Winograd transformation matrices introduce quantization errors when the transformations are performed in low-precision formats. Additionally, the transformed input tile values ($B^T a^{l-1} B$) are linear combinations of activations, causing the numerical range to expand by a factor referred to as the *enlargement factor* γ.
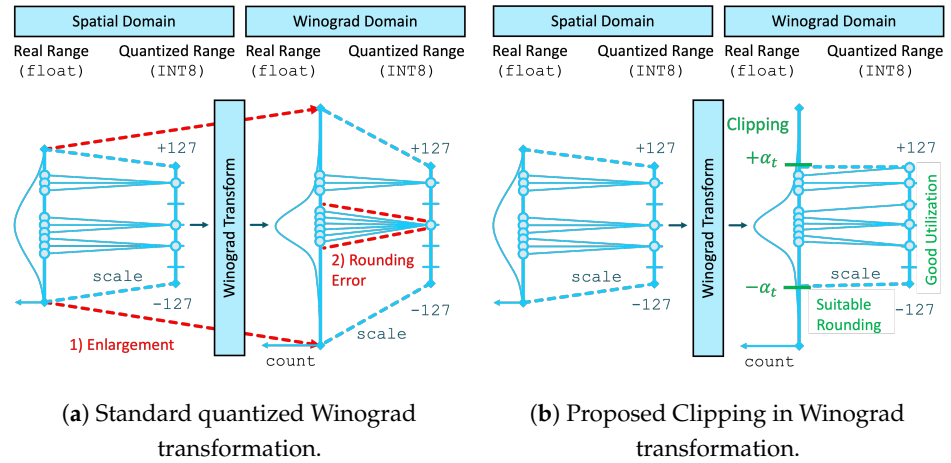
(**a**) Standard quantized Winograd transformation.

(**b**) Proposed Clipping in Winograd transformation.

Figure 2. Comparison of the (**a**) standard Winograd quantized transformation against (**b**) the Winograd quantized transformation that leverages trainable clipping factors to better exploit the quantized range.

This work considers 2D Winograd algorithms for convolution kernels of size 3×3, which are common in modern CNNs. Therefore, from now on, $F(m, 3)$ will be used to refer to the $F(m_x \times m_y, 3 \times 3)$ Winograd algorithm with $k_x = k_y = 3$. The operation reduction ratio, S, is related to the output tile size m as shown in Equation (7). The larger the size of m is, the more MAC reductions can be achieved with respect to standard convolution.

$$S = \frac{m^2 \times 3^2}{(m + 3 - 1)^2} \tag{7}$$

In Table 1, four different Winograd algorithms, namely $F(2, 3)$, $F(3, 3)$, $F(4, 3)$, and $F(6, 3)$, are compared. For each Winograd variant, the table reports the theoretical reduction in the number of operations (S) with respect to standard convolution and the maximum enlargement factor (γ) computed considering the worst-case scenario. Moreover, according to the SotA [23,24], this work considers that the weight transformation ($Gw^l G^T$) is performed offline, using floating-point representation to avoid unnecessary transform latency and quantization error at run-time.

Table 1. Comparison of different Winograd algorithms against standard convolution in terms of enlargement factor (γ), weight memory, theoretical and practical (ResNet-18) number of MAC savings (#MAC).

Algorithm	γ	Weight Memory	#MAC Reduction Theoretical (S)	ResNet-18
F(2,3)	4×	1.78×	2.25×	1.76×
F(3,3)	36×	2.78×	3.24×	2.05×
F(4,3)	**100×**	**4×**	**4×**	**2.45×**
F(6,3)	156.25×	7.1×	5.06×	2.24×

3.3. Clipping Factors in the Winograd Domain

Our latest work [21] investigates the Winograd algorithms to fully understand the main challenges that prevent their deployment on edge hardware. A key focus of the analysis is the distribution of numerical values for quantized layers, specifically regarding weights and activations, both before and after the application of the Winograd transformation. One significant observation is that the data range for activations expands following the transformation due to the enlargement factor discussed in Section 3.2. Upon analyzing the distribution of transformed activations and weights, it is observed that 99.9% of the data fall within a small portion of the total range, while the rest remain sparsely populated

(Figure 7). When quantizing this extensive floating-point range to an 8-bit format, it can be noticed a notable underutilization of the available 2^8 discrete quantized values. This is amplified by the fact that small magnitude values in the floating-point representation often map to the same or very similar values in the quantized representation, leading to a high incidence of rounding errors and further underutilization of the quantization range. This phenomenon is depicted in Figure 2a. To address this issue, the concept of **clipping factors** ($\boldsymbol{\alpha_t}$) is introduced within the Winograd domain. The transformed range is clipped to α_t according to layer statistics; in particular, $[-\alpha_t, +\alpha_t]$ is defined as the range that contains 99.9% of the available data distribution. Weights and activations have independent clipping factors (i.e., α_{ta} and α_{tw}).

With this approach, the transformed inputs and weights better exploit the available discrete, quantized values (Figure 2b).

The algorithm in Equation (4) can be reformulated as in Equation (8), where $\mathtt{Q_{Cw}}$ denotes the clipping in the Winograd domain.

$$a^l = A^T \left[\mathtt{Q_{Cw}}(G\mathtt{Q}(w^l)G^T) \odot \mathtt{Q_{Cw}}(B^T \mathtt{Q_C}(a^{l-1})B) \right] A \tag{8}$$

3.3.1. Trainable Clipping Factors α_{ta}, α_{tw}

While the introduction of clipping factors represents a significant step toward more effectively utilizing the quantized range, relying on static and manually defined values for these factors is inherently suboptimal. To address this limitation, a more adaptive strategy is proposed, allowing the clipping factors, α_{ta} for activations and α_{tw} for weights, to be dynamically adjusted during training through gradient descent with the aim of minimizing the numerical error brought by Winograd transformations and quantization. Firstly, the clipping function is applied to each value x_f belonging to the floating point range, limiting the distribution range to $[-\alpha_t, +\alpha_t]$:

$$x_c = 0.5(|x_f + \alpha_t| - |x_f - \alpha_t|) = \begin{cases} -\alpha_t, & x_f \in (-\infty, -\alpha_t) \\ x_f, & x_f \in [-\alpha_t, +\alpha_t) \\ +\alpha_t, & x_f \in [+\alpha_t, +\infty) \end{cases} \tag{9}$$

Then, x_c is quantized linearly on the available 8-bits quantized range, obtaining the discrete value x_q:

$$x_q = \mathtt{Round}\left(x_c \cdot \frac{2^{(8-1)} - 1}{\alpha_t} \right) \tag{10}$$

During backpropagation, the gradient $\frac{\delta x_q}{\delta \alpha_t}$ is evaluated using STE [28] to approximate $\frac{\delta x_q}{\delta x_f} = 1$:

$$\frac{\delta x_q}{\delta \alpha_t} = \frac{\delta x_q}{\delta x_f}\frac{\delta x_f}{\delta \alpha_t} = \begin{cases} -1, & x_f \in (-\infty, -\alpha_t) \\ 0, & x_f \in [-\alpha, +\alpha_t) \\ +1, & x_f \in [+\alpha_t, +\infty) \end{cases} \tag{11}$$

In summary, each layer of a convolutional neural network (CNN) model utilizes three distinct scalar clipping factors: c, which is employed for clipping input activations as described in [5]; α_{ta}, which is utilized for clipping transformed activations within the Winograd domain; and α_{tw}, which serves to limit the range of transformed weights in the Winograd domain, as illustrated in Figure 3. In terms of hardware, the approach requires **only two scalar** scaling factors per **layer**: one for the activation transform and one for standard output quantization (since weights are quantized offline), resulting in a negligible computation overhead. Initialization of clipping factors is a crucial operation that affects the prediction quality during training. During the warm-up phase of training, the data distribution is carefully assessed to select initial α values such that 99.9% of the transformed data values (transformed weights and activations) fall within the range

$[-\alpha_t, +\alpha_t]$. Additionally, L2 regulatization is not used, since it tends to penalize high-magnitude values indiscriminately, rather than focusing on their impact on accuracy.
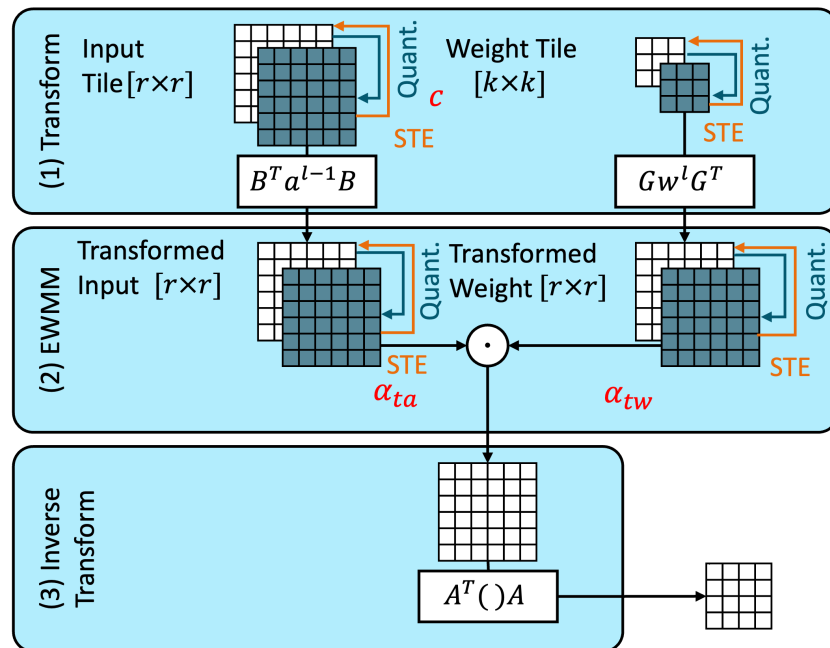

Figure 3. Overview of the proposed Winograd aware quantized training. Straight-through estimator (STE) is used to approximate the gradient of the quantization function. Trainable clipping factors c, α_{ta}, and α_{tw} are highlighted in red.

3.4. Winograd F(4,3) Convolution on GPU

In this section, the CUDA kernels designed to implement the 8-bit quantized Winograd F(4,3) convolutional algorithm with clipping factor presented in the previous section are described. A dedicated kernel has been designed for each step of the Winograd algorithm (input transformation, EWMM and output transformation), ensuring synchronization by having each kernel wait for the previous one to complete before starting (non-fused approach). This approach takes advantage of the large memory bandwidth and maximizes resource utilization for each kernel while maintaining correct execution order across the steps. Each kernel is organized in thread blocks and within each block the computation is parallelized over the input channel dimension ($P_{ic} < C_i$), the output channel dimension ($P_{oc} < C_o$), and the spatial dimension of the tile. Differently from other approaches, the design proposed in this work is optimized for batch size 1, targeting edge scenarios. *Weight transformation* is performed offline in floating-point precision; the resulting weights in the Winograd domain are then quantized on 8-bit W_w, minimizing the quantization error.

3.4.1. Input Transform

The input transformation kernel takes care of transforming the 8-bit quantized input tensor $\mathcal{A}^{l-1} \in \mathbb{R}^{H_i \times W_i \times C_i}$ to the Winograd domain and applying the quantization $\mathtt{Q_{Cw}}$ of Equation (8). The thread blocks are arranged in a 2D array ($N_{tiles} \times C_{tiles}$) as depicted in Figure 4, where N_{tiles} represents the number of tiles needed to produce the output spatial dimension and C_{tiles} refers to the number of tiles needed to transform all the C_i channels, P_{ic} at a time, computed as reported in Equation (12) and Equation (13), respectively. First, each thread block ($TB_{i,j}$) fetches the 8-bit pixels belonging to the corresponding tile obtaining a $6 \times 6 \times P_{ic}$ sub-volume (T_{in}). At this point, zero-padding (p) is added to the tiles belonging to the edges of the input feature map to match the expected output feature size. In some cases, extra padding is applied to the right and bottom edges to guarantee an integer number of 6×6 tiles in the spatial dimensions (yellow tile in Figure 4). Then, T_{in} is transformed to the Winograd domain throughout the matrix multiplications $B^T T_{in} B$ that

are fully unrolled and performed on integer arithmetic. The resulting 16-bit, transformed tile (T_{inW}) is *clipped* to the desired range ($[-\alpha_t, +\alpha_t]$), *quantized* on 8-bit (Section 3.3.1), and stored back in DRAM.

$$N_{tiles} = \left\lceil \frac{H_o}{4} \right\rceil * \left\lceil \frac{W_o}{4} \right\rceil \tag{12}$$

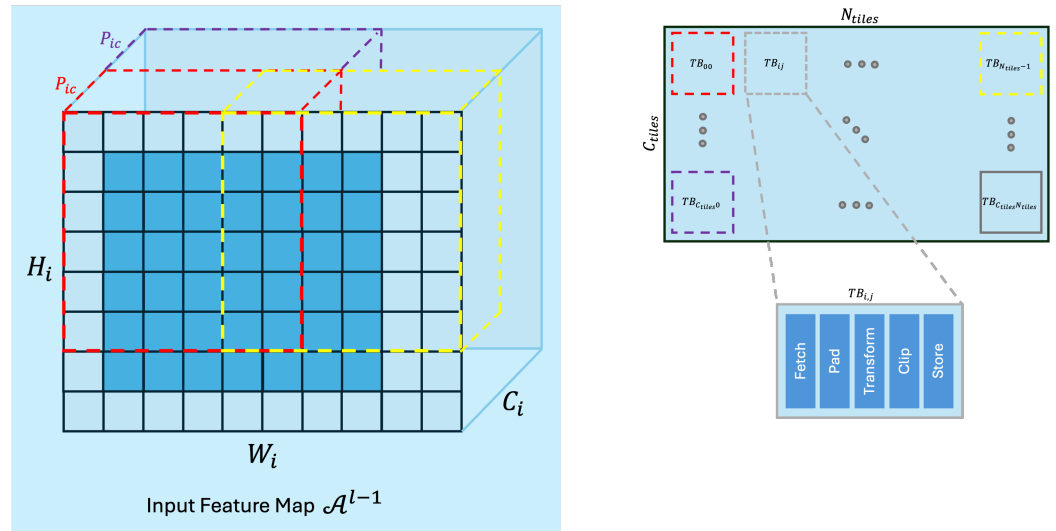$$C_{tiles} = \left\lceil \frac{C_i}{P_{ic}} \right\rceil \tag{13}$$

Figure 4. Input transformation kernel overview. The input volume is divided in sub-volumes and each thread block is responsible for the transformation of a sub-volume.

3.4.2. Element-Wise Matrix Multiplication

The core computation of the Winograd algorithm is represented by the element-wise matrix multiplication (EWMM). Here, the transformed weights (W_w) and transformed activations (T_{inW}) are multiplied point-wise to compute the convolutional result in the Winograd domain. In Section 3.2, the 2D Winograd algorithm for a 2D input a^{l-1} a 2D weight w^l is described. In CNN layers, the resulting tiles after the inverse transformation ($A^T(\cdot)A$) are accumulated over the input channel dimension (C_i) to produce the convolutional output in the spatial domain (a^l). However, the accumulation can be moved to the Winograd domain, right after the EWMM in order to reduce the computation of the inverse transformation. The $F(4, 3)$ Winograd convolutional algorithm for a 4×4 output tile a^l can be reformulated as in Equation (14).

$$a^l = \sum_{i=0}^{C_i} A^T[(Gw_i^l G^T) \odot (B^T a_i^{l-1} B)]A = A^T \left[\sum_{i=0}^{C_i} [(Gw_i^l G^T) \odot (B^T a_i^{l-1} B)] \right] A \tag{14}$$

Therefore, the EWMM kernel performs the point-wise multiplication of transformed weights and activation *and* accumulate results over the channel dimension C_i. The EWMM has been reformulated as a 6×6 GEMMs array (Figure 5). The kernel receives as inputs the 8-bit transformed input tiles $T_{inW} \in \mathbb{R}^{N_{tiles} \times C_i \times 6 \times 6}$ and the 8-bit transformed weights $W_w \in \mathbb{R}^{C_o \times C_i \times 6 \times 6}$ and produces the output tiles $T_{ewmm} \in \mathbb{R}^{N_{tiles} \times C_o \times 6 \times 6}$ that will be fed to the inverse transform kernel. Each GEMM ($GEMM_{i,j}$) in the 6×6 grid is responsible for all the computation involving the spatial tile coordinates (i, j) of all the input T_{inW} and all the weights W_w tiles. In detail, the $GEMM_{i,j}$ performs the matrix multiplication of the activation matrix Mat_A of size $N_{tiles} \times C_i$ and the weight matrix Mat_W of the dimension in $C_i \times C_o$ resulting in the output matrix Mat_O of the dimension in $N_{tiles} \times C_o$. Moreover, the highly optimized GEMM kernel (*cublasGemmEx()*) from cuBLAS [29] has been adopted in order to

enforce the 8-bit quantized EMWW to run on the fast and efficient Tensor Cores [30]. At the end of the computation, the $N_{tiles} \times C_o \times 6 \times 6$ tiles are written back to DRAM.
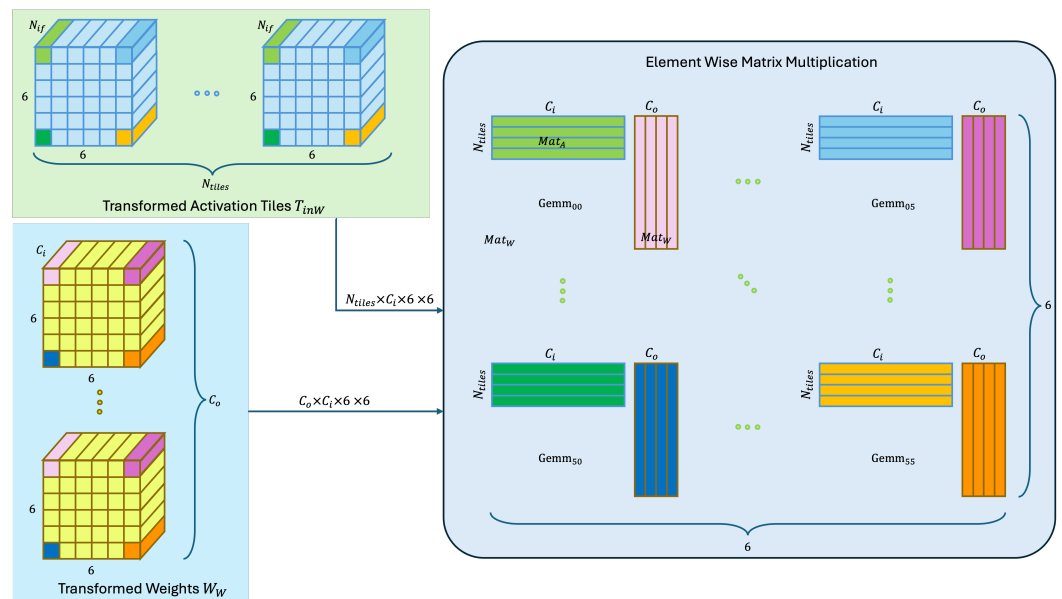

Figure 5. Element-wise matrix multiplication kernel overview. The computation is organized in 6×6 GEMMs. Each one is responsible for the computation of $N_{tiles} \times C_o$ output pixels in the Winograd domain.

3.4.3. Output Transform

The output transformation kernel is in charge of transforming the $N_{tiles} \times C_o \times 6 \times 6$ tiles coming from the EWMM kernel back to the spatial domain, producing the 8-bit convolutional output $\mathcal{A}^l \in \mathbb{R}^{H_o \times W_o \times C_o}$ (Figure 6). Also, in this case, the thread blocks are arranged in a 2D array ($N_{tiles} \times C_{tiles}$). N_{tiles} represents the number of tiles produced by the input transformation and EWMM kernels. C_{tiles} refers to the number of tiles needed to transform all the C_o channels, P_{oc}, at a time (Equation (15)).

$$C_{tiles} = \left\lceil \frac{C_o}{P_{oc}} \right\rceil \tag{15}$$

Each thread block ($TB_{i,j}$) fetches an integer tile $P_{oc} \times 6 \times 6$ and transforms it back to the spatial domain throughout the matrix multiplications $A^T(\cdot)A$ that are fully unrolled. The resulting 4×4 tile is clipped to the desired range ($[-c, +c]$) and quantized on 8-bit. At this point, the resulting tile is ready to be written back to DRAM. In case H_o or W_o are not divisible by 4, the 4×4 tiles belonging to the right and bottom edges are resized, cutting out the extra values in order to match the expected output spatial dimensions (gray pixels in the yellow tile in Figure 6).
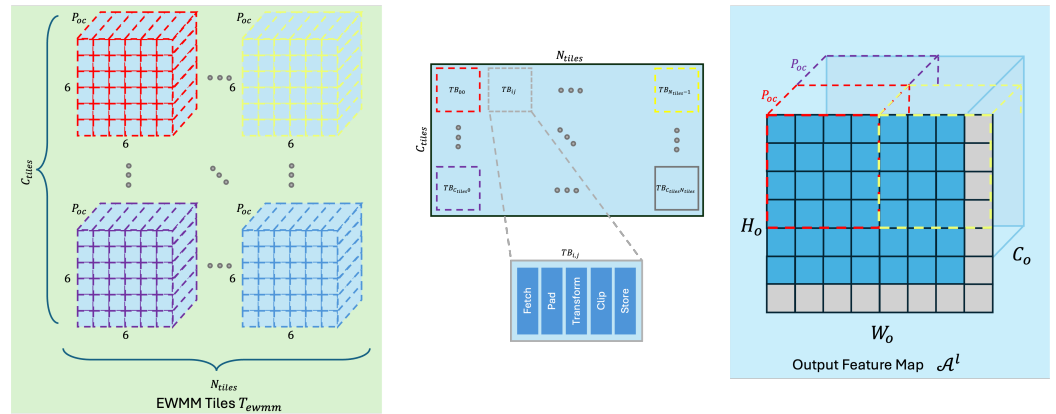
Figure 6. Inverse transformation kernel overview. The Winograd tiles produced by the EWMM kernel are transformed back to the spatial domain. Each thread block is responsible for the computation of a $4 \times 4 \times P_{oc}$ output pixel.

4. Experiments

This section presents the results of the proposed Winograd-aware 8-bit quantized training (Section 3.3) on various models (ResNet-20, ResNet-18 [1], VGG-9 [31], and DeepLabv3+ [3]) and on different datasets (CIFAR-10 [32], ImageNet [33], and CityScapes [34]). Unless otherwise specified, all training hyperparameters are adopted from the base implementation. Additionally, the effectiveness of the 8-bit quantized $F(4, 3)$ CUDA kernel presented in Section 3.4 is evaluated on the NVIDIA Jetson Orin Nano 8 GB in the 7 W configuration with frequency locked to 408 MHz, Jetpack 5 SDK [35] and CUDA 11. In all the experiments, $P_{ic} = P_{oc} = 128$.

4.1. Quantized Winograd with Clipping Factors

In Table 2, the effectiveness of the proposed approach is evaluated on different models (ResNet-20, VGG-9, ResNet-18, and DeepLabV3+) and datasets (CIFAR-10, ImageNet, and Cityscapes). For each experiment, the number of bits for weights and activations ($N_{W/A}$), the accuracy, and the overall MAC reduction achieved through Winograd are reported. The terms Conv and QConv refer to the standard convolutional algorithm implemented in the floating-point and quantized domains, respectively, while WinoQ-Conv denotes experiments where the quantized Winograd algorithm is applied to the QConv-restored checkpoint.

In the ResNet-20-CIFAR-10 experiments, the effectiveness of the trainable clipping approach is first evaluated against a post-training quantized network. The standard 8-bit quantized model (QConv) is restored, and the Winograd $F(4, 3)$ algorithm is enabled in the forward pass (WinoQConv). Model weights are kept fixed, while only the clipping factors for each layer and batch normalization parameters are calibrated. Although some accuracy degradation remains, clipping improves prediction quality by +46.75 percentage points compared to the standard PTQ Winograd $F(4, 3)$ algorithm (WinoQConv). To further reduce the accuracy gap, WAT is then enabled. Vanilla WAT alone achieves 89.69% accuracy and improves to 90.89% with the approach with trainable clipping factors.

In the VGG-9-CIFAR-10 experiments, Winograd-aware training with learnable clipping factors improves WinoQConv accuracy by +3.32 p.p.

The effectiveness of the proposed approach was subsequently evaluated using the ResNet-18 [1] model on the more complex, large-scale ImageNet [33] dataset. In all experiments, the model was trained for 80 epochs using a cosine learning rate decay schedule, starting at 0.08 and decaying to 0.0. The results consistently demonstrated the benefits of introducing trainable clipping factors in the Winograd domain, resulting in notable improvements in prediction accuracy. Specifically, for the Winograd-aware model trained with $F(4, 3)$, an enhancement of +3.43 percentage points in prediction accuracy was achieved by incorporating trainable clipping parameters.

Table 2. Influence of the proposed Winograd quantize training on ResNet-20 and VGG-9 on CIFAR-10, ResNet-18 on Imagenet and DeepLabV3+ on Cityscapes.

Dataset	Model	Method	$N_{W/A}$	QAT/ WAT	Algorithm	Winograd Clipping	Saving	Top-1 [%]
Cifar-10 [32]	ResNet-20 [1]	Conv [1]	32	✗	-	-	-	91.61
		QConv [5]	8	✓	-	-	-	91.39
		WinoQConv	8	✗	F(4,3)	✗	3.4×	35.36
		Ours	8	✗	F(4,3)	✓	3.4×	82.11
				✓	F(4,3)	✗	3.4×	89.69
				✓	F(4,3)	✓	**3.4×**	**90.89**
	VGG-9 [31]	QConv	8	✓	-	-	-	93.11
		WinoQConv	8	✓	F(4,3)	✗	3.84×	88.97
		Ours	8	✓	F(4,3)	✓	3.84×	**92.29**
Imagenet [33]	ResNet-18 [1]	Conv [1]	32	✗	-	-	-	71.00
		QConv [5]	8	✓	-	-	-	70.54
		WinoQConv	8	✗	F(4,3)	✗	2.45×	5.45
		Ours	8	✓	F(4,3)	✗	2.45×	65.71
				✓	F(4,3)	✓	**2.45×**	**69.14**
CityScapes [34]	DeepLabV3+ [3]	QConv [5]	8	✓	-	-	-	67.82
		Ours	8	✓	F(4,3)	✓	2.56×	66.57

In the final set of experiments, the analysis was extended to a semantic segmentation task. The DeepLabv3+ [3] architecture was used on the CityScapes [34] dataset, with a modified ResNet-18 backbone in which the last two residual blocks were removed. Once again, the Winograd-aware approach with trainable clipping factors effectively reduced accuracy degradation compared to standard quantized convolution, confirming its robustness in preserving model performance even in segmentation tasks.

4.2. Effect of Clipping Factors

This section demonstrates how the proposed method effectively constrains the range of the transformed weights and activations, resulting in improved quantization performance. Figure 7 presents the distribution of the Winograd $F(4, 3)$ transformed weights and activations for three specific layers (layers 15, 16, and 17) of the ResNet-20 model, trained on the CIFAR-10 dataset. The x-axis shows the unique values of the transformed weights and activations, while the y-axis (on a logarithmic scale) reflects the frequency of occurrences within each respective layer. By limiting the range of these transformed values, our approach reduces outliers in the Winograd domain, leading to a smoother distribution that is more suitable for quantization.

For the activations, the enlargement factor increases the numerical range in the Winograd domain, causing a huge quantization error that leads to severe accuracy degradation. The approach proposed, mitigates this issue by dynamically constraining the distribution of activations, ensuring more efficient use of the available quantized range. Additionally, our method proves highly effective in clipping the transformed weights, preserving over 99% of the values that appear in the Winograd transform. By applying trainable clipping, it is ensured that only a minimal portion of the weight distribution falls outside the clipped range (red area), while the core data necessary for storing the information are retained (green area).
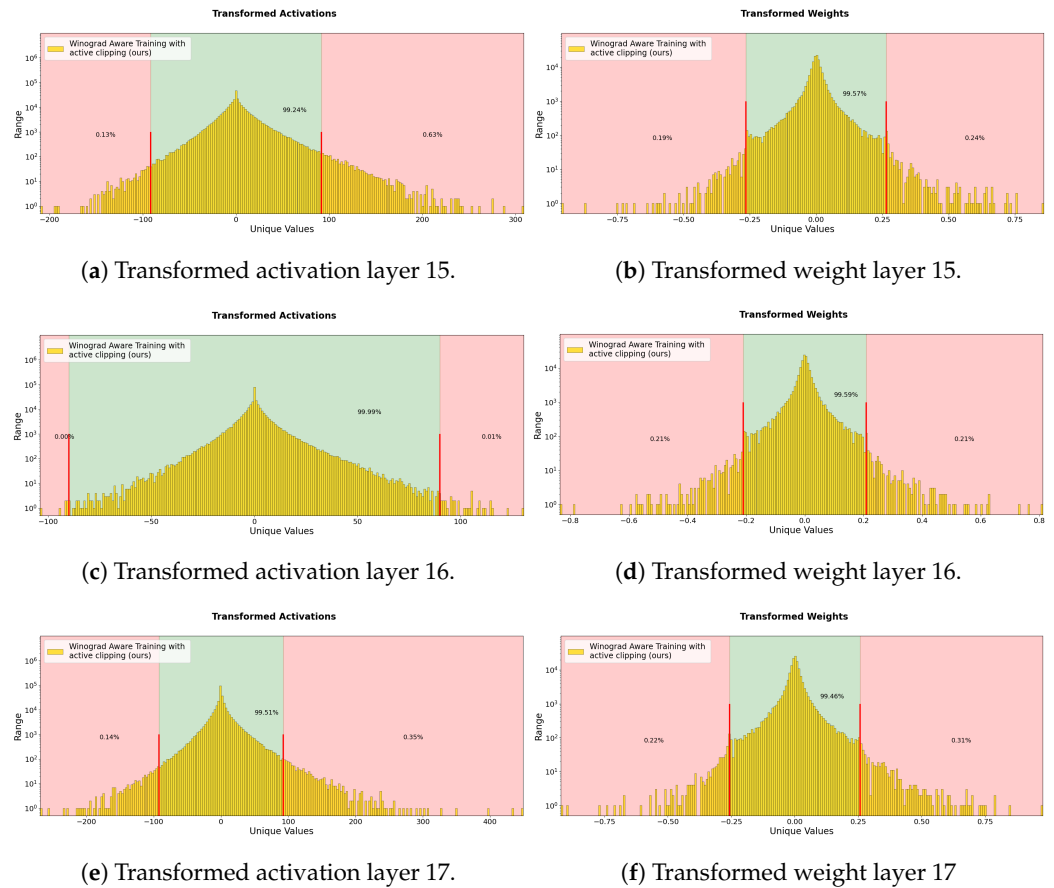
(**a**) Transformed activation layer 15.

(**b**) Transformed weight layer 15.

(**c**) Transformed activation layer 16.

(**d**) Transformed weight layer 16.

(**e**) Transformed activation layer 17.

(**f**) Transformed weight layer 17

Figure 7. Numerical distributions of example layers for transformed weights and activations of ResNet-20 on CIFAR-10. The values in the clipped range (green) sufficiently contain the information needed to maintain high-accuracy full 8-bit Winograd.

4.3. Winograd GPU Kernel Speedup

In Figure 8, the latency speedup achieved by the custom Winograd kernel described in Section 3.4 is reported in comparison to cuDNN's best-performing algorithm, which leverages 8-bit quantization and Tensor Cores (the `cudnnConvolutionForward()` function with the `CUDNN_CONVOLUTION_ FWD_ALGO_IMPLICIT_PRECOMP_GEMM` algorithm), referred to as `int8x32`. The speedup was evaluated across multiple layers with varying spatial dimensions H_o and W_o (ranging from 32×32 to 256×256) and channel dimensions C_i and C_o (256, 512, and 1024). For small spatial and channel dimensions ($H_o \times W_o < 64 \times 64$ and $C_i = C_o = 256$) the standard convolutional algorithm performs better than the Winograd kernel because of the limited number of spatial tiles in the Winograd domain that causes an underutilization of the hardware. However, for larger spatial and/or channel dimensions, the proposed kernel outperforms the standard convolutional algorithm by up to $3.41\times$. It is interesting to observe that the speedup brought by a larger channel dimension is much higher than the one brought by a larger spatial dimension.

4.4. Contribution of Each Step to the Latency

We show in Figure 9 the latency contribution of each step of the Winograd algorithm (input transformation, element-wise matrix multiplication and inverse transformation) on the total latency. For each set of spatial dimensions (H_o and W_o), the contribution of each step is presented across multiple channel dimensions (C_i, C_o). For small spatial dimensions (32×32) most of the latency is taken by the EWMM. Interestingly, the contributions of the input and inverse transformations increase with larger spatial dimensions. This is caused by two reasons: first, the number of tiles (N_{tiles}) increases with the spatial dimensions, leading to more input and inverse transformations to be performed. Second, the size of the matrices

involved in the EWMM gets higher with the increase in the number of tiles, leading to a better utilization, reducing the contribution of the EWMM to the overall latency.
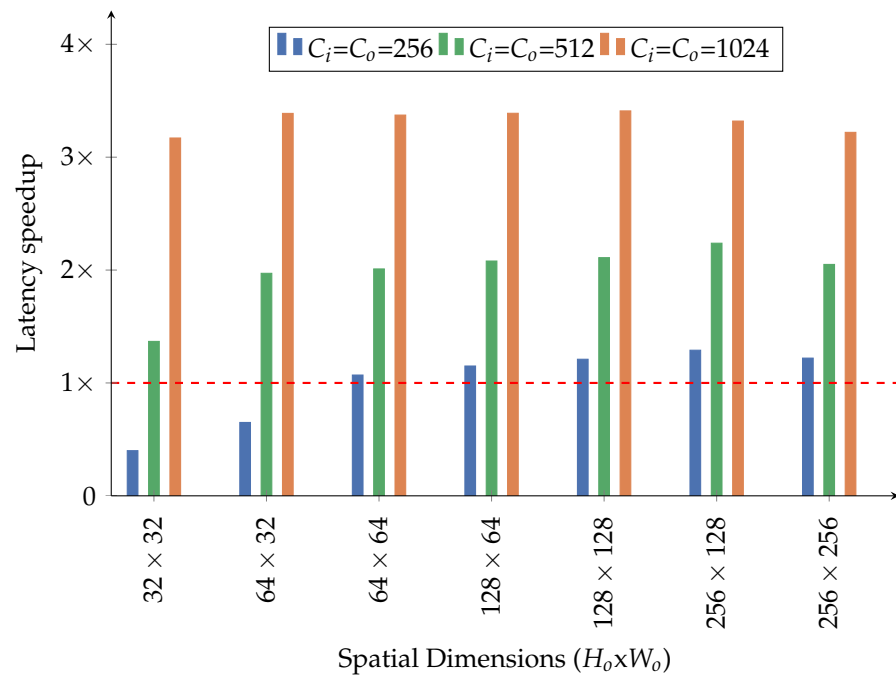

Figure 8. Latency speedup brought by the custom Winograd $F(4, 3)$ kernels compared to cuDNN convolution on Tensor Cores (`int8x32`).
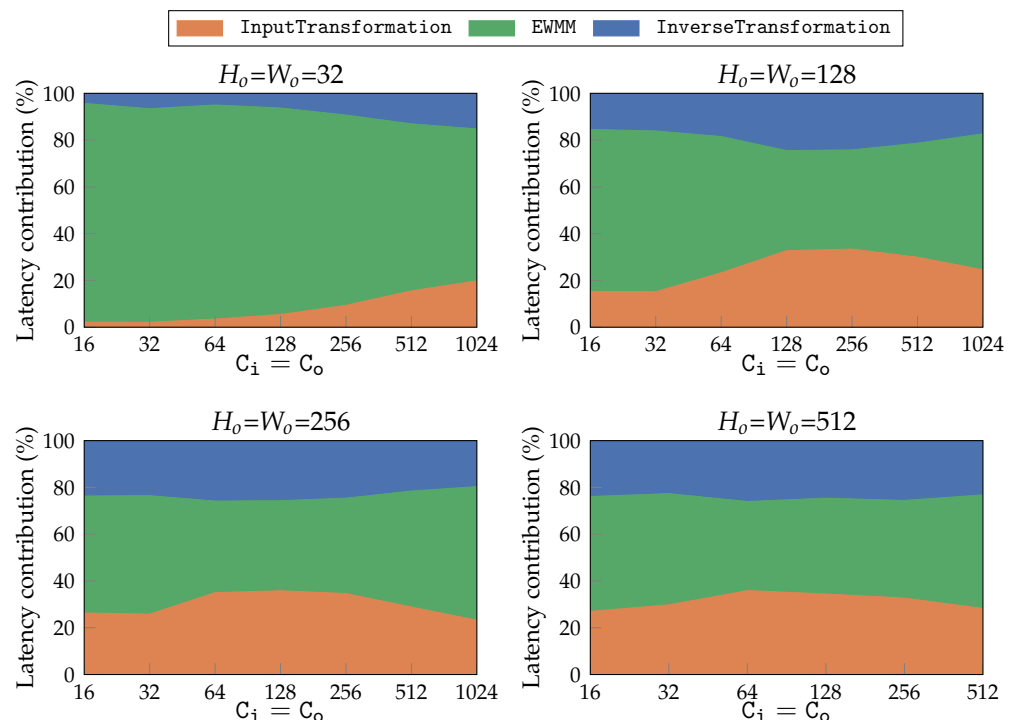

Figure 9. The latency contribution of each of the three steps in the Winograd $F(4, 3)$ algorithm. In each sub-figure, the spatial dimensions are fixed, while the channel dimensions are varied.

4.5. Layer-Wise Latency Comparison

In this final experiment, the ResNet-18 [1] backbone of the DeepLabV3+ [3] model is considered. Table 3 presents a layer-wise latency comparison between the proposed

custom kernel, the cuDNN-optimized implementation of convolution utilizing the DP4A instruction on CUDA cores (`int8x4`), and Tensor Cores (`int8x32`, Section 4.3).

For each layer, the configuration (C_i, C_o, W_o, H_o), the latency of the three implementations, and the speedup with respect to the `int8x32` implementation are reported. In all cases, the Winograd kernel proposed in this work outperforms the standard convolution implementation based on the DP4A instruction. When compared against the `int8x32` variant, our kernel shows a lower latency for deeper layers, characterized by larger channel dimensions, achieving a speedup of $2.1\times$ for the last layers.

Table 3. Latency comparison of different layers in the proposed Winograd kernel against cuDNN's `int8x4` and `int8x32` variants. The speedup is calculated relative to the `int8x32` variant.

| Name | Layer Config | | | | Latency [ms] | | | | Speedup |
	C_i	C_o	W_o	H_o	int8x4	int8x32	**Ours**	**Best**	[$\times$]
conv2_block1_2	64	64	512	256	25.84	5.08	13.76	5.08	1.00
conv2_block2_1	64	64	512	256	25.84	5.08	13.76	5.08	1.00
conv2_block2_2	64	64	512	256	25.84	5.08	13.76	5.08	1.00
conv3_block1_2	128	128	256	128	25.13	4.79	7.05	4.79	1.00
conv3_block2_1	128	128	256	128	25.13	4.79	7.05	4.79	1.00
conv3_block2_2	128	128	256	128	25.13	4.79	7.05	4.79	1.00
conv4_block1_2	256	256	128	64	24.80	4.57	**3.96**	3.96	**1.15**
conv4_block2_1	256	256	128	64	24.80	4.57	**3.96**	3.96	**1.15**
conv4_block2_2	256	256	128	64	24.80	4.57	**3.96**	3.96	**1.15**
conv5_block1_1	256	512	128	64	49.54	9.07	**6.22**	6.22	**1.46**
conv5_block1_2	512	512	128	64	98.33	17.82	**8.49**	8.49	**2.10**
conv5_block2_1	512	512	128	64	98.33	17.82	**8.49**	8.49	**2.10**
conv5_block2_2	512	512	128	64	98.33	17.82	**8.49**	8.49	**2.10**
Total					571.84	105.85	106.00	**73.18**	**1.44**

Although some layers cannot take advantage of the proposed Winograd kernel, in Table 3, it is possible to observe how both implementations (ours and the `int8x32`) result in a similar total latency. It is also interesting to observe how for each layer the best kernel can be selected at compile time based on the layer configuration. In the reported example, when the `int8x32` implementation is selected for the first layers and ours for the remaining ones, the overall latency goes down to 73.18 ms, bringing a $1.44\times$ reduction to the overall latency.

5. Conclusions and Discussion

In this work, we proposed a method to tackle the numerical instability that affects the 8-bit Winograd $F(4,3)$ algorithm on edge devices. We demonstrated how using trainable clipping factors for transformed weights and transformed activations in the Winograd domain, the data range can be limited, reducing the quantization error and providing a better mapping to the 8-bit quantized range. With the proposed training scheme we achieve $2.45\times$ and $2.56\times$ MAC reduction with minimal degradation in prediction quality for ResNet-18-ImageNet and DeepLabV3+ on CityScapes, respectively. Furthermore, we designed a novel kernel for accelerating the proposed Winograd quantized algorithm on edge GPUs, taking advantage of both 8-bit quantization and Tensor Cores. We showed how our Winograd kernel can reduce the latency of the standard convolution algorithm on edge GPUs by $3.41\times$ compared to the optimized cuDNN implementation. Finally, we showed how our kernel can be used combined with the standard cuDNN implementations to maximize the benefits on edge GPUs. In future work, we aim to extend the benefits of our approach across different hardware platforms and explore multiple Winograd variants to further generalize its application.

Data Availability Statement: The original contributions presented in this study are included in the article material. Further inquiries can be directed to the corresponding authors.

Conflicts of Interest: Authors Pierpaolo Mori, Mohammad Shanur Rahman, Lukas Frickenstein, Shambhavi Balamuthu Sampath, Moritz Thoma, Nael Fasfous, Manoj Rohit Vemparala and Alexander Frickenstein were employed by the company BMW AG. The remaining authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

CNN	Convolutional Neural Network
EWMM	Element-Wise Matrix Multiplication
PTQ	Post-Training Quantization
QAT	Quantization-Aware Training
MAC	Multiply and Accumulate
WAT	Winograd-Aware Training
FP	Floating Point
STE	Straight-Through Estimator
GEMM	General Matrix Multiply

References

1. He, K.; Zhang, X.; Ren, S.; Sun, J. Deep residual learning for image recognition. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, Las Vegas, NV, USA, 27–30 June 2016; pp. 770–778.
2. Zhou, X.; Wang, D.; Krähenbühl, P. Objects as Points. *arXiv* **2019**, arXiv:1904.07850.
3. Chen, L.C.; Zhu, Y.; Papandreou, G.; Schroff, F.; Adam, H. Encoder-decoder with atrous separable convolution for semantic image segmentation. In Proceedings of the European Conference on Computer Vision (ECCV), Munich, Germany, 8–14 September 2018; pp. 801–818.
4. Implementing the Tensorflow Deep Learning Framework on Qualcomm's Low-Power DSP. 2020. Available online: https://www.edge-ai-vision.com/2017/07/implementing-the-tensorflow-deep-learning-framework-on-qualcomms-low-power-dsp-a-presentation-from-google/ (accessed on 23 October 2024).
5. Choi, J.; Wang, Z.; Venkataramani, S.; Chuang, P.I.J.; Srinivasan, V.; Gopalakrishnan, K. Pact: Parameterized clipping activation for quantized neural networks. *arXiv* **2018**, arXiv:1805.06085.
6. Zhou, S.; Wu, Y.; Ni, Z.; Zhou, X.; Wen, H.; Zou, Y. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv* **2016**, arXiv:1606.06160.
7. Khalil, K.; Eldash, O.; Kumar, A.; Bayoumi, M. Designing Novel AAD Pooling in Hardware for a Convolutional Neural Network Accelerator. *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* **2022**, *30*, 303–314. [CrossRef]
8. Nvidia Jetson Nano Developer KIT. 2019. Available online: https://cdn.sparkfun.com/assets/0/7/f/9/d/jetson-nano-devkit-datasheet-updates-us-v3.pdf (accessed on 17 July 2024).
9. Snapdragon 8 Gen 1 Mobile Platform. 2021. Available online: https://www.qualcomm.com/content/dam/qcomm-martech/dm-assets/documents/snapdragon-8-gen-1-mobile-platform-product-brief.pdf (accessed on 23 October 2024).
10. Google Coral. 2020. Available online: https://coral.ai/static/files/Coral-M2-Dual-EdgeTPU-datasheet.pdf (accessed on 23 October 2024).
11. Lavin, A.; Gray, S. Fast algorithms for convolutional neural networks. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, Las Vegas, NV, USA, 27–30 June 2016; pp. 4013–4021.
12. Maji, P.; Mundy, A.; Dasika, G.; Beu, J.; Mattina, M.; Mullins, R. Efficient Winograd or Cook-Toom Convolution Kernel Implementation on Widely Used Mobile CPUs. *arXiv* **2019**, arXiv:1903.01521.
13. Alam, S.A.; Anderson, A.; Barabasz, B.; Gregg, D. Winograd Convolution for Deep Neural Networks: Efficient Point Selection. *arXiv* **2022**, arXiv:2201.10369. [CrossRef]

14. Barabasz, B.; Anderson, A.; Soodhalter, K.M.; Gregg, D. Error analysis and improving the accuracy of Winograd convolution for deep neural networks. *ACM Trans. Math. Softw. (TOMS)* **2020**, *46*, 1–33. [CrossRef]

15. Kim, M.; Park, C.; Kim, S.; Hong, T.; Ro, W.W. Efficient Dilated-Winograd Convolutional Neural Networks. In Proceedings of the 2019 IEEE International Conference on Image Processing (ICIP), Taipei, Taiwan, 22–25 September 2019; pp. 2711–2715. [CrossRef]

16. Jiang, J.; Chen, X.; Tsui, C.Y. A Reconfigurable Winograd CNN Accelerator with Nesting Decomposition Algorithm for Computing Convolution with Large Filters. *arXiv* **2021**, arXiv:2102.13272.

17. Yang, C.; Wang, Y.; Wang, X.; Geng, L. WRA: A 2.2-to-6.3 TOPS highly unified dynamically reconfigurable accelerator using a novel Winograd decomposition algorithm for convolutional neural networks. *IEEE Trans. Circuits Syst. I Regul. Pap.* **2019**, *66*, 3480–3493. [CrossRef]

18. Liu, X.; Pool, J.; Han, S.; Dally, W.J. Efficient Sparse-Winograd Convolutional Neural Networks. *arXiv* **2018**, arXiv:1802.06367.

19. Yang, T.; Liao, Y.; Shi, J.; Liang, Y.; Jing, N.; Jiang, L. A Winograd-Based CNN Accelerator with a Fine-Grained Regular Sparsity Pattern. In Proceedings of the 2020 30th International Conference on Field-Programmable Logic and Applications (FPL), Gothenburg, Sweden, 31 August–4 September 2020; pp. 254–261. [CrossRef]

20. Fernandez-Marques, J.; Whatmough, P.; Mundy, A.; Mattina, M. Searching for Winograd-aware Quantized Networks. In Proceedings of the Machine Learning and Systems, Austin, TX, USA, 2–4 March 2020; Dhillon, I., Papailiopoulos, D., Sze, V., Eds.; Volume 2, pp. 14–29.

21. Mori, P.; Frickenstein, L.; Sampath, S.B.; Thoma, M.; Fasfous, N.; Vemparala, M.R.; Frickenstein, A.; Unger, C.; Stechele, W.; Mueller-Gritschneder, D.; et al. Wino Vidi Vici: Conquering Numerical Instability of 8-Bit Winograd Convolution for Accurate Inference Acceleration on Edge. In Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision (WACV), Waikoloa, HI, USA, 3–8 January 2024; pp. 53–62.

22. Li, G.; Jia, Z.; Feng, X.; Wang, Y. Lowino: Towards efficient low-precision winograd convolutions on modern cpus. In Proceedings of the 50th International Conference on Parallel Processing, Lemont, IL, USA, 9–12 August 2021; pp. 1–11.

23. Chikin, V.; Kryzhanovskiy, V. Channel Balancing for Accurate Quantization of Winograd Convolutions. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), New Orleans, LA, USA, 18–24 June 2022.

24. Barabasz, B. Quantaized winograd/toom-cook convolution for dnns: Beyond canonical polynomials base. *arXiv* **2020**, arXiv:2004.11077.

25. Andri, R.; Bussolino, B.; Cipolletta, A.; Cavigelli, L.; Wang, Z. Going Further with Winograd Convolutions: Tap-Wise Quantization for Efficient Inference on 4×4 Tiles. In Proceedings of the 2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO), Chicago, IL, USA, 1–5 October 2022; pp. 582–598.

26. Castro, R.L.; Andrade, D.; Fraguela, B.B. OpenCNN: A Winograd Minimal Filtering Algorithm Implementation in CUDA. *Mathematics* **2021**, *9*, 2033. [CrossRef]

27. Liu, J.; Yang, D.; Lai, J. Optimizing Winograd-Based Convolution with Tensor Cores. In Proceedings of the 50th International Conference on Parallel Processing, New York, NY, USA, 9–12 August 2021; ICPP'21. [CrossRef]

28. Bengio, Y.; Léonard, N.; Courville, A. Estimating or propagating gradients through stochastic neurons for conditional computation. *arXiv* **2013**, arXiv:1308.3432.

29. Nvidia. cuBLAS. 2019. Available online: https://docs.nvidia.com/cuda/cublas/index.html (accessed on 23 October 2024).

30. Nvidia. Tensor Cores. 2019. Available online: https://www.nvidia.com/en-us/data-center/tensor-cores/ (accessed on 23 October 2024).

31. Simonyan, K.; Zisserman, A. Very Deep Convolutional Networks for Large-Scale Image Recognition. *arXiv* **2015**, arXiv:1409.1556.

32. Krizhevsky, A.; Hinton, G. *Learning Multiple Layers of Features from Tiny Images*; University of Toronto: Toronto, ON, Canada, 2009.

33. Russakovsky, O.; Deng, J.; Su, H.; Krause, J.; Satheesh, S.; Ma, S.; Huang, Z.; Karpathy, A.; Khosla, A.; Bernstein, M.; et al. ImageNet Large Scale Visual Recognition Challenge. *Int. J. Comput. Vis. (IJCV)* **2015**, *115*, 211–252. [CrossRef]

34. Cordts, M.; Omran, M.; Ramos, S.; Scharwächter, T.; Enzweiler, M.; Benenson, R.; Franke, U.; Roth, S.; Schiele, B. The Cityscapes Dataset. In Proceedings of the CVPR Workshop on the Future of Datasets in Vision, Boston, MA, USA, 7–12 June 2015.

35. Nvidia. Jetpack. 2024. Available online: https://developer.nvidia.com/embedded/jetpack (accessed on 4 November 2024).