SEARCHING FOR WINOGRAD-AWARE QUANTIZED NETWORKS

Javier Fernandez-Marques 1  Paul N. Whatmough 2  Andrew Mundy 2  Matthew Mattina 2

ABSTRACT
Lightweight architectural designs of Convolutional Neural Networks (CNNs) together with quantization have paved the way for the deployment of demanding computer vision applications on mobile devices. Parallel to this, alternative formulations to the convolution operation such as FFT, Strassen and Winograd, have been adapted for use in CNNs offering further speedups. Winograd convolutions are the fastest known algorithm for spatially small convolutions, but exploiting their full potential comes with the burden of numerical error, rendering them unusable in quantized contexts. In this work we propose a Winograd-aware formulation of convolution layers which exposes the numerical inaccuracies introduced by the Winograd transformations to the learning of the model parameters, enabling the design of competitive quantized models without impacting model size. We also address the source of the numerical error and propose a relaxation on the form of the transformation matrices, resulting in up to 10% higher classification accuracy on CIFAR-10. Finally, we propose wiNAS, a neural architecture search (NAS) framework that jointly optimizes a given macro-architecture for accuracy and latency leveraging Winograd-aware layers. A Winograd-aware ResNet-18 optimized with wiNAS for CIFAR-10 results in 2.66× speedup compared to im2row, one of the most widely used optimized convolution implementations, with no loss in accuracy.

1 INTRODUCTION
The rise in popularity of deep CNNs has spawned a research effort to find lower complexity networks to increase inference efficiency. This is desirable for inference in the cloud and becomes crucial on mobile and IoT devices with much more constrained hardware (Lane & Warden, 2018). Over the last few years, multiple approaches have been proposed to alleviate the compute-bound nature of convolutions (Sze et al., 2017). Arguably, the use of depthwise convolutions, as popularized by the family of MobileNet architectures (Howard et al., 2017; Sandler et al., 2018; Howard et al., 2019), has become the most widely embraced design choice to make lightweight networks. These layers are used in state of the art image classification networks (Stamoulis et al., 2019; Tan & Le, 2019). However, beyond image classification, normal convolutions are still chosen in favour of depthwise convolutions for applications like image super-resolution (Zhang et al., 2018; Lee et al., 2019), image segmentation (Takikawa et al., 2019) and, GANs (Brock et al., 2019). Therefore, alternative forms of speeding up standard convolutions are required to run these applications in mobile CPUs, which often come with constrained compute and energy budgets (Whatmough et al., 2019). Model quantization and the use of alternative convolution algorithms instead of direct convolution are two ways of accomplishing this task.

Lower-precision networks result in smaller model sizes, faster inference, lower energy consumption and smaller chip area (Sze et al., 2017; Li et al., 2019). Concretely, 8-bit quantitized models achieve comparable performance to full-precision models (Jacob et al., 2018; Krishnamoorthi, 2018) while being ready for deployment on off-the-shelf hardware as 8-bit arithmetic is widely supported. In addition to resulting in a direct 4× model size reduction, 8-bit integer-only arithmetic benefits from up to 116× and 27.5× chip area reduction compared to full precision additions and multiplies respectively, requiring 30× and 18.5× less energy (Horowitz, 2014; Whatmough et al., 2018). Because of these desirable benefits, 8-bit quantization has been widely adopted in both compute-constrained devices (Liberis & Lane, 2019; Chowdhery et al., 2019; Wang et al., 2019) and accelerators (Whatmough et al., 2019).

Orthogonal to lightweight architectural designs and quantization, fast convolution algorithms in replacement of direct
convolutions can provide further speedups. These come with their own trade-offs (Anderson & Gregg, 2018), but in this work we focus on the Winograd algorithm since it is the fastest known algorithm for convolutions of the dimensions often found in CNNs. The Winograd convolution performs the bulk of the computation as a Hadamard product between weights and input in the Winograd space requiring $O(n)$ operations. Unlike normal convolutions, that generate a single output per convolution, a Winograd convolution computes several outputs simultaneously. This property makes Winograd convolutions minimal in the number of general multiplications\(^1\) (Winograd, 1980). Normal convolutions operate on tile sizes matching the width and height of the filter, on the other hand, Winograd convolutions can operate on larger tiles. This is illustrated in Figure 1. In this way, while normal convolutions would require 8.1K multiplications to densely convolve a $32 \times 32$ input with a $3 \times 3$ filter, Winograd convolutions operating on a $4 \times 4$ tile require only 3.6K. The speedups Winograd convolutions offer increase with tile size. However, exploiting these speedups exposes a problem inherent in current Winograd convolution implementations: numerical error. This error, which grows exponentially with tile size, is the primary reason why Winograd is generally only deployed with 32-bit floating point and for comparatively small tile sizes, rarely larger than $6 \times 6$. In practice, an architecture with standard convolutional layers would first be trained before replacing standard convolution with Winograd convolution for deployment.

In this paper, we focus on alleviating the problem of numerical error that arises when using Winograd convolutions in quantized neural networks. Achieving this ultimately enables us to combine the speedups of Winograd with those that reduced precision arithmetic is known to offer, among other benefits in terms of energy and area. To this end, we present an end-to-end training pipeline that exposes the numerical inaccuracies introduced by Winograd to the learning of the model parameters. We also address the source of the numerical error and propose a relaxation on the form of the transformation matrices used in the Winograd convolution algorithm. We achieve this by adding these matrices to the set of learnable parameters in a layer, after initializing them via Cook-Toom (L. Toom, 1963). Finally, we describe wiNAS, a Winograd-aware Neural Architecture Search framework which leverages Winograd-aware layers and latency measurements on Arm Cortex-A73 and A53 cores, to jointly optimize for high accuracy and low latency. Our framework transforms a given macro-architecture by replacing each convolution with either im2row or Winograd convolutions of different tile sizes.

\footnotetext{\(\text{General multiplications}\) is a term commonly used in Winograd jargon referring to element-wise or Hadamard product stage.}

The contributions of this work are summarized below:

- We show that Winograd-aware networks enable Winograd convolutions in quantized networks, including 8-bit networks with little accuracy drop. To the best of our knowledge, this is the first time this has been empirically demonstrated.
- We demonstrate that learning the Winograd transforms, as opposed to keeping these fixed, results in better network accuracy – up to 10% improvement when using $6 \times 6$ and $8 \times 8$ tiles in 8-bits CNNs with $3 \times 3$ filters. This improvement is more pronounced with larger $5 \times 5$ filters.
- We present wiNAS as a tool that can find Winograd-aware networks jointly optimised for both high accuracy and low latency given a real hardware model.

2 RELATED WORK

Convolutions have become the de facto spatial feature extractor in neural networks. As a result, a number of approaches have emerged to reduce the computational costs of using this operator.

Compact CNN Architectures. These include alternative formulations to the dense convolutional layer, such as bottleneck layers (He et al., 2016) that perform the $3 \times 3$ convolutions in a lower-dimensional space, or the depth-wise convolutional layers (Howard et al., 2017) which replace the standard convolutions with a channel-wise convolution followed by a $1 \times 1$ point-wise convolution. More recently, Chen et al. (2019) proposed a compact multi-resolution convolutional block that reduces spatial redundancy of low frequencies resulting in faster inference, memory savings and slightly higher accuracy. This reinforces the proposition that current networks rely more on texture than shape for image/object discrimination (Brendel & Bethge, 2019; Geirhos et al., 2019). In this work, instead of presenting a new architecture, we propose an optimization for an existing known-good architecture to speed up inference. Our

[Figure 1. Standard convolutions operate on a tile (blue) defined by the filter size, generating a single output. Winograd convolutions operate on larger tiles without modifying the filter dimensions. A $3 \times 3$ Winograd convolution operating on a $4 \times 4$ tile (red) generates a $2 \times 2$ output. This is often expressed as $F(2 \times 2, 3 \times 3)$.]

\(^1\) We present wiNAS as a tool that can find Winograd-aware networks jointly optimised for both high accuracy and low latency given a real hardware model.
optimization can be applied to existing pre-trained models without the need for end-to-end training.

**Quantization.** The most extreme form of quantization is binary networks (Courbariaux et al., 2015; Lin et al., 2017; Xiang et al., 2017;), which replace convolutions with bit-shifts resulting in $58 \times$ inference speed-ups (Rastegari et al., 2016). Ternary and 2-bit models (Li et al., 2016; Wan et al., 2018; Gong et al., 2019) achieve higher accuracies while alleviating some the challenges of training binary networks (Alizadeh et al., 2019). However, it is 8-bit quantization (Jacob et al., 2018; Krishnamoorthi, 2018; Wang et al., 2019) that has achieved high popularity due to its balance between accuracy, model size reduction and inference speedup. Newer data formats, such as Posit (Carmichael et al., 2019) aim to close the accuracy gap between INT8 and FP32 networks, however hardware supporting it is unavailable. For training, BFLOAT16 (Kalamkar et al., 2019) has been validated as an alternative to FP32, enabling faster training. In this work, we adopt INT8 and INT16 uniform quantization during training and study how lowering precision impacts on the lossy nature of Winograd convolutions.

**Fast Convolution Algorithms.** Alternative formulations of the convolution operation such as: the use of FFTs, which replace convolution with its multiplication-only counterpart in the frequency domain resulting in faster inference (Mathieu et al., 2013; Abtahi et al., 2018) and training (Highlander & Rodriguez, 2015); the Strassen algorithm (Strassen, 1969), which when applied to convolutions (Cong & Xiao, 2014; Tscharnlen et al., 2018) significantly reduces the number of multiplications at the cost of more additions; or the Winograd algorithm (Winograd, 1980), which replaces convolutions with a set of matrix transformations and point-wise multiplications and, results in significantly faster inference stages (Lavin & Gray, 2016).

**Winograd Convolution.** The Winograd algorithm for fast convolution was first applied to CNNs by Lavin & Gray (2016), showing $2.2 \times$ speedup compared to cuDNN (Chetlur et al., 2014) on a VGG (Simonyan & Zisserman, 2015) network, with no loss in accuracy on small $4 \times 4$ tiles, batch 1. However, exploiting Winograd convolutions on larger input tiles is challenging due to numerical instability. In response to this limitation, Barabasz et al. (2018) showed that the error introduced by the Winograd algorithm grows at least exponentially with tile size, which can be partially alleviated by choosing better polynomial points for constructing the transformation matrices via Cook-Toom (L. Toom, 1963). An alternative formulation using trimmed Vandermonde matrices was described by Vincent et al. (2017). More recently, several works studying the suitability of Winograd convolutions in memory and compute constrained setups have been proposed. These include: the use of integer arithmetic for complex Winograd convolutions (Meng & Brothers, 2019); a general formulation for the Winograd algorithm (Barabasz & Gregg, 2019) that shows promising results in FP16 and BFLOAT16 when using higher degree polynomials; an efficient region-wise multi-channel implementation of Winograd convolutions using General Matrix Multiplications (GEMMs) (Maji et al., 2019) that achieves $4 \times$ speedups on Arm Cortex-A CPUs; and, a technique (Liu et al., 2018) that enables up to 90% sparsity in the Hadamard product stage of the Winograd algorithm, effectively reducing by $10 \times$ the number of multiplications with no accuracy loss in FP32 models. Our work fills the gap of using Winograd convolutions in quantized neural networks, enabling even faster convolutions in current off-the-shelf hardware, such as mobile CPUs.

**Neural Architecture Search.** Automating the process of designing neural network architectures has drawn considerable attention. Early attempts relied on reinforcement learning (Zoph & Le, 2017; Brock et al., 2018; Real et al., 2019; Tan et al., 2019) or Bayesian optimization (Hernández-Lobato et al., 2016; Fedorov et al., 2019) and required thousands of GPU hours to converge due to their computationally expensive and exhaustive search stages. Other works opted instead for a gradient-based search by framing the problem as a single over-parameterized network where all candidate operations at a particular node (e.g. a layer) are taken into consideration. The main aspect differentiating gradient-based NAS approaches is the way the output of a layer combines the contribution of each candidate operation. While Bender et al. (2018) defines it as the sum and DARTS (Liu et al., 2019) as a weighted sum, ProxylessNAS (Cai et al., 2019) relies on path-level binarization, making it possible to perform the search on the entire architecture directly using a single GPU. In addition to architecture discovery, NAS has also been successfully used for automated network pruning (He et al., 2018) and quantization (Wang et al., 2019). Our work leverages NAS to find the optimal convolution algorithm (i.e. im2row or different Winograd implementations) for each layer in the model while preserving the overall network macro-architecture and model size.

3 **WINOGRAD-AWARE NETWORKS**

This section introduces Winograd convolutions and their trade-offs in terms of compute, memory and accuracy. Then, we present the Winograd-aware layers used in our networks.

3.1 **Winograd implementation trade-offs**

The Winograd algorithm for convolutions using linear polynomials guarantees to use the minimum number of element-wise multiplications to compute $m \times m$ outputs using an $r \times r$ filter. Lavin & Gray (2016) refer to this minimal
algorithm as $F(m \times m, r \times r)$ and present its matrix form:

$$Y = A^T \left[ (GgG^T) \odot (B^T dB) \right] A$$

where $G, B$ and $A$ are transformation matrices applied to the filter $g$, input and output respectively and $\odot$ is the Hadamard or element-wise multiplication.

These transformation matrices are commonly constructed as described in the Cook-Toom algorithm which requires choosing a set of so-called polynomial points from $\mathbb{R}^2$. This choice is not trivial, but for small Winograd kernels e.g. $F(2 \times 2, 3 \times 3)$ or $F(4 \times 4, 3 \times 3)$, there is a common consensus. While a standard convolution using a $r \times r$ filter $g$ would operate on a $r \times r$ input tile, a Winograd convolution expressed as Eq. 1 expects an input patch $(m + r - 1) \times (m + r - 1)$. The key difference is that while the standard convolution would generate a $1 \times 1$ output, the Winograd convolution would compute a $m \times m$ output. In this way, a standard $3 \times 3$ convolution requires 9 multi- per output (mpo), $F(2 \times 2, 3 \times 3)$ and $F(4 \times 4, 3 \times 3)$ require 4 mpo and 2.25 mpo respectively. Theoretically, these savings grow as we increase the tile or filter sizes. For the remaining of this work and, unless stated otherwise, we will be considering $3 \times 3$ filters and therefore refer to $F(2 \times 2, 3 \times 3)$ as $F2$, $F(4 \times 4, 3 \times 3)$ as $F4$, and so on.

The challenges associated with the use of Winograd convolutions span three dimensions:

**Compute.** Winograd convolutions require the transformation of both tile $d$ and filter $g$ to the Winograd domain. The cost of these transformations grows with $m$, and can represent a significant portion of the total computation of up to 75% (Sec. 6.2). This suggests that Winograd offers little to no speedup in layers with few filters. The cost of $GgG^T$ is often ignored as it is amortized across inferences.

**Memory.** In Eq.1, $GgG^T$ transforms the filter $g$ to the Winograd domain, matching the dimensions of the input tile $d$. This results in an increase of run-time memory associated with the weights: $1.78 \times 4 \times 4$ for $F2$ and $F4$ respectively. This is especially undesirable on memory-constrained devices such as microcontrollers.

**Numerical Error.** Small $F2$ and $F4$ perform well in single and double precision (FP32/64) networks and are available in production-ready libraries such as NVIDIA cuDNN (Chetlur et al., 2014) and Arm Compute Library (Arm Software). Because these introduce only marginal numerical error, a network can first be trained using conventional convolutions before replacing appropriate layers with Winograd, without impacting accuracy.

3See Section 5.2 in Blahut (2010) for a step-by-step example.

<table>
<thead>
<tr>
<th>Convolution method</th>
<th>ResNet-18 Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct</td>
<td>93.16</td>
</tr>
<tr>
<td>Winograd F2</td>
<td>93.16</td>
</tr>
<tr>
<td>Winograd F4</td>
<td>93.14</td>
</tr>
<tr>
<td>Winograd F6</td>
<td>93.11</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Convolution method</th>
<th>ResNet-18 Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct</td>
<td>93.60</td>
</tr>
<tr>
<td>Winograd F2</td>
<td>93.48</td>
</tr>
<tr>
<td>Winograd F4</td>
<td>19.25</td>
</tr>
<tr>
<td>Winograd F6</td>
<td>11.41</td>
</tr>
</tbody>
</table>

Table 1. Replacing the convolutional layers in pre-trained ResNet-18 models on CIFAR-10 with $F2$, $F4$ and $F6$. This works well in full precision, but accuracy drops drastically with quantization for configurations beyond $F2$. Note that prior to evaluating the quantized configurations we performed a warmup of all the moving averages involved in Eq.1 using the training set but without modifying the weights. Without this relaxation (which requires a Winograd-aware pipeline as in Fig. 2), $F2$ would be unusable.

However, attempting this with larger Winograd tiles, or in combination with quantization, results significant accuracy loss. The root of the problem is the increasing numerical range in $G, B$ and $A$ as $d$ increases. As a consequence, the multiple matrix multiplications in Eq.1 contribute considerable error, ultimately reducing accuracy. This problem is exacerbated in networks using quantized weights and activations, where the range and precision of values is reduced. We show these limitations in Table 1. The numerical error is, to a large extent, the main limiting factor for adopting large-tile Winograd and for adopting Winograd convolutions in general for reduced precision networks.

In this work we focus on minimizing the numerical errors that arise when using the Winograd algorithm in quantized networks. Our approach does not aggravate the compute and memory challenges previously mentioned. Instead, it indirectly alleviates these by making use of quantization.

### 3.2 A Winograd-aware training pipeline

Neural networks have proven to be resilient to all kinds of approximations, e.g. pruning and quantization. When applying these techniques, consistently better models are generated if these approximations are present during training. In other words, when the training is aware of quantization, or when training is aware of pruning.

Following this intuition, we propose an end-to-end Winograd-aware pipeline as shown in Figure 2. In the forward pass we apply Eq.1 to each patch of the activations from the previous layer. We can apply standard back-propagation, since Eq.1 is only a collection of matrix-matrix multiplications. This implementation allows us to:

- **Learn better filters.** Building an explicit implementation of each of the stages involved in the Winograd
transform exposes the numerical errors introduced in Eq. 1 to the learning of the filters. This prevents the accuracy drops shown in Table 1.

- **Learn the transforms.** Traditionally, matrices $G$, $B^\top$ and $A^\top$ are fixed. Instead, we can treat them as another set of learnable parameters in the layer. This relaxation leads to much improved performance in quantized networks while still maintaining the overall structure of the Winograd convolution algorithm and its speedups.

- **Quantization diversity.** Unlike standard convolution, which does not require intermediate computation, Winograd convolution requires at least four of them for $GgG^\top$, $B^\top dB$, the Hadamard product and the output transformation. Each of these can be quantized to a different number of bits depending on the bit-width of the input, that of the weights, and the overall complexity of the problem the network is designed to solve.

4 SEARCHING FOR WINOGRAD-AWARE NETWORKS

Simultaneously maximizing accuracy and minimizing latency with Winograd convolution isn’t trivial. The reason for this is that large tiles result in low latency, but come at the cost of higher numerical error. This presents a good opportunity to jointly optimize network accuracy and latency.

To this end, we implement a NAS-based approach that automatically transforms an existing architecture into a Winograd-aware version. We perform NAS at the micro-architecture level by selecting from different convolution algorithms for each layer, but without modifying the network’s macro-architecture (e.g. number or order of layers, hyper-parameters, etc). Keeping the macro-architecture fixed allows us to fairly compare the standard model to its Winograd-aware counterpart in terms of latency and accuracy. We call our framework wiNAS.

4.1 Winograd-aware NAS pipeline

Introducing latency measurements into the optimization objective requires knowing the shape of the input tensor, i.e. the activations from the previous layer, at each layer of the network. We design wiNAS as a variation of ProxylessNAS (Cai et al., 2019), leveraging path sampling while performing the search. This technique, enables the allocation of the entire network on a single GPU by evaluating no more than two candidate operations at each layer per batch.

Similarly to ProxylessNAS, wiNAS formulates the search as a two-stage process, alternating the update of model parameters (the weights), where the loss is defined as

$$L_{weights} = Loss_{CE} + \lambda_0 \| w \|_2^2$$

(2)

and the update of architecture parameters (the weight assigned to each operation on a given layer), where the loss introduces the latency metrics is defined as

$$L_{arch} = Loss_{CE} + \lambda_1 \| a \|_2^2 + \lambda_2 E\{latency\}$$

(3)
Winograd convolutions. For each experiment with Winograd convolutional layers. Therefore, the set of candidate operations varying tile size. While the former is lossless and faster than direct convolution, Winograd offers lower latencies but introduce numerical instability that could ultimately impact in accuracy.

where \( a \) are the architecture parameters and \( \lambda_2 \) controls the impact of latency in the loss. The expected latency, \( E(\text{latency}) \), for a given layer is the weighted combination of the latency estimate of each candidate operation with their respective probability of being sampled. Intuitively, searching for Winograd convolutions with high \( \lambda_2 \) would result in faster models, potentially at the detriment of accuracy.

Unlike ProxylessNAS, wiNAS focuses on simply selecting the optimal convolution algorithm for each of the \( 3 \times 3 \) convolutional layers. Therefore, the set of candidate operations for a given \texttt{conv2d} layer contains \texttt{im2row} and Winograd-aware layers in their \texttt{F2}, \texttt{F4} and \texttt{F6} configurations. This search space is illustrated in Figure 3. Each candidate operation comes with its respective latency, which is a function of the output dimensions and quantization level.

5 EXPERIMENTAL SETUP

We conduct various experiments grouped in three categories. In this section we describe each experiment we conducted. We used PyTorch (Paszke et al., 2017) for training and Arm Compute Library for deployment.

5.1 Vanilla Winograd-aware networks

We begin our study of Winograd-aware networks by performing an extensive evaluation on the ResNet-18 (He et al., 2016) architecture using the CIFAR-10 (Krizhevsky et al., 2009) dataset. In this experiment we train the network end-to-end using standard convolutions, and \texttt{F2}, \texttt{F4} and \texttt{F6} Winograd convolutions. For each experiment with Winograd, all layers in the network use the same tile size, except the last two residual blocks which are kept fixed to \texttt{F2}. The input convolutional layer uses normal convolutions. We run the experiments for FP32, INT16, INT10 and INT8 quantized networks, where both weights and activations are uniformly quantized (including all the intermediate outputs shown in Figure 2). We follow the per-layer symmetric quantization as described in Krishnamoorthi (2018). We repeated each experiment while enabling the Winograd transforms \( G, B^\top \) and \( A^\top \) to be learnt, which we denote using the additional suffix -flex.

Winograd-aware layers do not require an over-parameterized model to perform well. We also varied the model size by using a width-multiplier, as used by the MobileNets family, ranging from 0.125 to 1.0, meaning that when the multiplier is 1.0 the network is the full ResNet-18. This leads to models ranging between 215K and 11M parameters. Winograd-aware layers with learnable transformations marginally increase (\(< 0.1\%\)) the model size, since the transforms themselves need to be saved for model deployment. We repeated the experiment for CIFAR-100 (Krizhevsky et al., 2009), but without varying the depth-multiplier. CIFAR-100 is considerably more challenging that CIFAR-10, as it is comprised of 100 classes with only 600 images per class.

Additionally, we use an INT8 LeNet (Lecun et al., 1998), trained on the MNIST dataset, to evaluate the suitability of Winograd-aware layers with learnable transforms for \( 5 \times 5 \) filters. This is a more challenging case than \( 3 \times 3 \) filters, because a larger tile size is required (defined by \( F(m \times m, r \times r) \)), with larger transformation matrices which require the choice of more good polynomial points.

For experiments on ResNet-18, we replace \( 2 \times 2 \) stride convolution layers with a \( 2 \times 2 \) max-pooling layer followed by a dense \( 3 \times 3 \) convolution layer. Altering the network in this way is necessary since there is no known equivalent for strided Winograd convolutions, which remains an open research question. This is a common strategy when evaluating Winograd (Liu et al., 2018; Choi et al., 2018). We also modified the number of output channels of the input layer from 64 to 32. We did this to reduce the memory peak during training. We use the Adam (Kingma & Ba, 2015) optimizer and train for 120 epochs. Both CIFAR-10/100 use the same ResNet-18 architecture, differing only in the number of outputs of the fully connected layer. Results for other architectures are shown in A.1.

5.2 wiNAS: Winograd-aware NAS

To evaluate wiNAS, we define two different sets of candidate operations. These spaces are: wiNAS\textsubscript{WA} and wiNAS\textsubscript{WAQ}, both allowing each \( 3 \times 3 \) convolutional layer to be implemented with either \texttt{im2row} or each of the Winograd configurations, \( F2, F4 \) or \( F6 \). The former uses a fixed bit-width for all elements in the architecture, while the latter introduces in the search space candidates of each operation quantized to FP32, INT16 and INT8.
The hyperparameters used for wiNAS are as follows: for the learning of model parameters we use mini-batch SGD with Nesterov momentum (Sutskever et al., 2013). In the stage where we update the architecture parameters we use instead Adam with the first momentum scaling, $\beta_1$, set to zero, so the optimizer only updates paths that have been sampled. For both stages we use Cosine Annealing (Loshchilov & Hutter, 2017) scheduling and a batch size of 64. We perform the search for 100 epochs in each search space at different $\lambda_2$ values ranging from 0.1 to 1e-3. Once the search is completed, we trained the architecture end-to-end with the same hyperparameters as the rest of winograd-aware networks.

5.3 Winograd convolutions on mobile CPUs

For our study, we chose Arm A73 and A53 cores on a Huawei HiKey 960 development board with the big.LITTLE CPU architecture. These cores are good candidates for validating the speedups that are achievable with Winograd convolutions in today’s off-the-shelf mobile hardware.

<table>
<thead>
<tr>
<th>CPU</th>
<th>Clock</th>
<th>L1</th>
<th>L2</th>
</tr>
</thead>
<tbody>
<tr>
<td>A73</td>
<td>2.4 GHz</td>
<td>64 KB</td>
<td>2048 KB</td>
</tr>
<tr>
<td>A53</td>
<td>1.8 GHz</td>
<td>32 KB</td>
<td>512 KB</td>
</tr>
</tbody>
</table>

Table 2. Key hardware specifications for the high-performance Cortex-A73 and the high-efficiency Cortex-A53 cores found on a HiKey 960 development board.

While both A73 and A53 are implemented as 16nm quad-core CPUs, the former is a high-performance processor and the latter implements a high-efficiency processor. In Table 2 we summarise the main differences between these CPUs. The memory bandwidth would be the primary factor that ultimately sets the upper limit to the speedup achievable by Winograd since it requires operating in larger tiles than direct convolution algorithms such as im2row or im2col.

In our study, we measured the time taken for $3 \times 3$ convolutions using im2row, im2col and each of the Winograd configurations ($F_2$, $F_4$, $F_6$) when varying output width/height (from $112 \times 112$ down to $2 \times 2$) and $inCh \rightarrow outCh$ (from $3 \rightarrow 32$ to $512 \rightarrow 512$). We performed the benchmark in controlled conditions and in single thread mode. Each combination was run five times with five seconds delay in between to prevent thermal throttling. We implemented Winograd convolutions using GEMMs (Maji et al. (2019)), and performed the same experiment separately on A73 and A53 for both FP32 and INT8. INT16 measurements are not currently supported in Arm Compute Library.

6 Experimental Results

The results of this work are arranged as three subsections. First, we show that winograd-aware networks can achieve high accuracy. Second, we present the results from our dense benchmark for winograd convolutions on mobile CPUs. Third, we show that wiNAS can jointly optimize a given macro-architecture for accuracy and latency.

6.1 Vanilla Winograd-aware networks

Figure 4 (left) shows Winograd-aware networks in FP32 perform as well as direct convolutions, with both fixed and learned (-flex) transformation matrices. With quantization (all other plots), winograd-aware layers are essential to enable the use of fast Winograd convolutions. This is not possible if switching to Winograd convolutions after training, as is commonly done in practice (see Table 1).

Furthermore, we show that learning the Winograd transforms (-flex) results in 10% and 5% better accuracies for $F_4$ and $F_6$ in INT8 scenarios. We argue that enabling this relaxation helps in easing the numerical instability inherent to Winograd convolutions, which is further exacerbated by quantization. The accuracy of Winograd-aware models scales linearly with network width, suggesting that these can be exploited in conjunction with architecture compression techniques such as channel pruning.

Results from LeNet ($5 \times 5$ filters), provides further evidence that larger tiles result in higher numerical error. In Figure 5, we show that even in relatively small datasets like MNIST, keeping the transformations $G$, $B^T$ and $A^T$ fixed, leads to poor results as the output tile size is increased. This difference is almost 47% in the case of $F(6 \times 6, 5 \times 5)$ layers, which uses $10 \times 10$ tiles.

Winograd-aware layers do not structurally modify the network architecture, since Winograd is just an algorithm to perform convolution. We demonstrate it is possible to transform a pre-trained model with standard convolution into its Winograd-aware counterpart within a few epochs. Concretely, in Figure 6 we show that an INT8 ResNet-18 $F_4$ can be adapted from a model of the same network that was trained end-to-end with standard convolutions in 20 epochs of retraining. This represents a $2.8 \times$ training time reduction for Winograd-aware models. This is only possible when allowing the transformation matrices to evolve during training. Adapting FP32 models can be done in a single epoch.

We believe both $F_4$ and $F_6$ performance could be raised with alternative quantization implementations, closing the accuracy gap with $F_2$ and direct convolutions.

6.2 Impact of Winograd on Latency

The speedups associated to with use of Winograd convolutions often only account for the point-wise stage while assuming negligible costs for the input, weights and output transformations. Furthermore, these also assume that the larger the input patch, $d$, the larger the speedup compared to...
normal convolutions. However, although these assumptions are true for large tensors, they are not necessarily true when working with tensors of the sizes often found in CNNs for image classification or object detection.

Figure 7 shows a small portion of the obtained latency measurements for our benchmark in FP32. An almost identical pattern appears when using 8-bit arithmetic. In Figure 8 we show the speedups that Winograd convolutions offer at different layers of a ResNet-18 network. Our observations can be summarized in three points:

**Input layers do not benefit from Winograd.** This is primarily because the matrices in the element-wise GEMM stage are not large enough to compensate for the costs of transforming the input and output patches (see Figure 5.3 and 8). They represent a significant portion (up to 65% and 75% respectively on the A73 and A53) of the total costs for convolving a RGB $32 \times 32$ input expanded to 32 channels. Similar ratios can be observed for other input sizes. In spite of this, this first layer accounts for a marginal portion of the total latency of the model, often below 1ms.

**Optimal m is a function of input width and height.** For an input with sufficient number of channels, e.g. 32 channels and up, we observe a consistent pattern alternating between $F4$ and $F6$ as the channel dimension of the output increase. This alternation comes as a result of the impossibility of subdividing the input into an integer number of $(m+r-1) \times (m+r-1)$ patches, and therefore having to waste calculations when operating around the matrix edges. This pattern is invariant to different $inCh \rightarrow outCh$ configurations and fades away as input dimensions exceed $40 \times 40$, where $F6$ consistently becomes the fastest.

**Winograd transforms are costly.** Excluding inputs with very few channels, the cost of performing the transformations to/from the Winograd domain can exceed 25% of the overall costs. These costs become negligible as the input width and height decrease, but the rate at which this happens also depends on the hardware. Our Winograd-aware
pipeline formulation doesn’t impose any constraints on how the transforms are learnt. This results in dense transforms (as opposed to the default transforms which contain zeros) and therefore applying them require additional compute. Table 3 includes this latency overhead in models making use of learned transforms. In Appendix A.2 we provide more details on how dense transforms impact overall latency.

On A53, the speedups from FP32 Winograd convolutions are smaller than On A73. We argue this comes as a result of the differences in the memory subsystem, limiting the lower-end CPU to efficiently operate with larger tensors. These speedups grow significantly when leveraging INT8 arithmetic, made possible by winograd-aware training. Concretely, INT8 Winograd increases the speedup on the A53 by a factor of almost 1.5× compared to Winograd in FP32, as shown in W\textsubscript{A-F\textsubscript{4}} configurations in Table 3 – at the cost of 1.1% accuracy in CIFAR-10. In the case of the more challenging CIFAR-100 dataset, the drop in accuracy is more severe. However, our W\textsubscript{A-F\textsubscript{2}} layers offer attractive speedups for INT8 with no drop in accuracy. We rely on wiNAS to minimize this degradation with small impact on latency.

6.3 wiNAS Networks

Choosing the convolution algorithm that minimizes overall network latency can be easily done by looking at the benchmark results. However, since the accuracy of Winograd convolutions degrade rapidly in reduced precision networks, selecting the fastest algorithm for each layer without sacrificing accuracy, is not straightforward.

When using WiNAS\textsubscript{WA}, values of λ\textsubscript{2} larger than 0.05 consistently result in models with the same layer configuration as those in W\textsubscript{A-F\textsubscript{2}} (described in section 5.1). When lowering the impact of latency in Eq. 3 loss function, we observed several F\textsubscript{4} Winograd-aware layers were replaced with either im2row or F\textsubscript{2}, at the cost of less than 9 ms latency increase in the worst case, an INT8 model on the A53 for CIFAR-100. These models resulted in similar accuracies in FP32 and reached 0.32% and 1.1% higher accuracies in INT8 for CIFAR-10 and CIFAR-100 respectively. Despite CIFAR-100 models sacrificing more latency in order to recover accuracy, they remained faster than W\textsubscript{A-F\textsubscript{2}} at INT8.

When introducing quantization in the search performed by WiNAS\textsubscript{WA-Q}, the accuracy gap is almost closed for CIFAR-10 and further reduced for CIFAR-100. This comes primar-
Searching for Winograd-aware Quantized Networks

<table>
<thead>
<tr>
<th>Conv. Type</th>
<th>Bits act. / param.</th>
<th>Accuracy (%)</th>
<th>Cortex-A53</th>
<th>Cotex-A73</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CIFAR-10 CIFAR-100</td>
<td>Latency (ms)</td>
<td>Speedup</td>
<td>Latency (ms)</td>
</tr>
<tr>
<td>im2row</td>
<td>93.16 74.62</td>
<td>118</td>
<td>-</td>
<td>85</td>
</tr>
<tr>
<td>im2col</td>
<td>93.16 74.62</td>
<td>156</td>
<td>0.76×</td>
<td>102</td>
</tr>
<tr>
<td>W₂</td>
<td>93.16 74.60</td>
<td>126</td>
<td>0.94×</td>
<td>56</td>
</tr>
<tr>
<td>W₄</td>
<td>93.14 74.53</td>
<td>97</td>
<td>1.22×</td>
<td>46</td>
</tr>
<tr>
<td>WA₂₄</td>
<td>93.46 74.69</td>
<td>126</td>
<td>0.94×</td>
<td>56</td>
</tr>
<tr>
<td>WA₄</td>
<td>93.54 74.98</td>
<td>122¹</td>
<td>0.92×</td>
<td>54¹</td>
</tr>
<tr>
<td>wiNASWA</td>
<td>93.35 74.71</td>
<td>123¹</td>
<td>0.96×</td>
<td>56¹</td>
</tr>
<tr>
<td>im2row</td>
<td>93.20 74.11</td>
<td>117</td>
<td>1.01×</td>
<td>54</td>
</tr>
<tr>
<td>im2col</td>
<td>93.20 74.11</td>
<td>124</td>
<td>0.95×</td>
<td>59</td>
</tr>
<tr>
<td>WA₂₄</td>
<td>93.72 73.71</td>
<td>91</td>
<td>1.30×</td>
<td>38</td>
</tr>
<tr>
<td>WA₄</td>
<td>92.46 72.38</td>
<td>82¹</td>
<td>1.44×</td>
<td>35¹</td>
</tr>
<tr>
<td>wiNASWA</td>
<td>92.71 73.42</td>
<td>88³ / 91¹</td>
<td>1.34× / 1.30×</td>
<td>35³ / 36¹</td>
</tr>
<tr>
<td>wiNASWA-Q</td>
<td>auto</td>
<td>72.88</td>
<td>74³ / 97¹</td>
<td>1.60× / 1.22×</td>
</tr>
</tbody>
</table>

Table 3. Performance in terms of accuracy and latency (ms) of ResNet-18 when convolutions are implemented with different algorithms and for different quantization levels. We show that Winograd-aware (WA₂₄) layers combine the speedups of Winograd convolutions with those of INT8 arithmetic, with little to no accuracy loss in some cases. This is not possible with existing Winograd (W₂₄) formulations. Latency is measured on Arm Cortex-A73 and A53 cores. For the last two rows, wiNAS found different optimizations for each dataset. We show latencies for CIFAR-10 on the left and CIFAR-100 on the right. Speedups are shown against im2row in FP32. (*) With default Winograd transforms. (†) Includes worst case latency increase due to using learned transform, which are often dense.

7 DISCUSSION

In this section we present some of the challenges of training Winograd-aware networks and propose lines of future work.

A direct implementation of Eq. 1 requires saving the intermediate outputs of each matrix-matrix multiplication, since these are needed for back-propagation. This results in high memory usage. In this work we had to rely on gradient checkpointing (Chen et al., 2016) to lower the memory peak during training, at the cost of additional computation. We believe a native CUDA implementation of the Winograd-aware layers with better memory reuse would ease this problem.

Learning larger models (with width multipliers 0.75 and 1.0) proved challenging for F4 and F6 when introducing quantization. Using other types of quantization would likely help. In particular per-channel affine quantization, as in Jacob et al. (2018). Also, enabling different bit-widths throughout Eq. 1 could help mitigate the accuracy drop.

It is known that bad polynomial points for constructing $G$, $Bᵀ$ and $Aᵀ$ introduce significant deviations in the result of computing Winograd convolutions compared to that of normal convolutions. We observed that good starting points are also important even when learning the Winograd transformations. Polynomial points specifically tailored for quantized Winograd could alleviate some of the degradation that we observed with increased tile size.

In this work we focused on mobile CPUs, but we expect these benefits to be also applicable to GPUs. However, to further maximize the speedups that Winograd-aware layers for quantized CNNs offer, a custom hardware implementation in the form of an accelerator would be preferable.

8 CONCLUSION

Running CNN-based applications that require standard convolutional layers is challenging in compute-constrained devices such as mobile CPUs. This paper presents Winograd-aware layers as the building block to combine the benefits of quantized networks and fast Winograd convolutions. We studied Winograd-aware layers with different tile sizes, three quantization levels and on three popular datasets. We found that allowing the transformation matrices to evolve during training resulted in significantly better models. With wiNAS we leveraged Winograd-aware layers and latency metrics from off-the-shelf mobile CPUs and found architectures that helped minize the numerical instability of Winograd. A Winograd-aware ResNet-18 quantized to INT8 offers up to 1.32× faster inference for only a marginal accuracy drop compared to existing Winograd implementations, which are limited to FP32. This network is also 1.54× faster than an optimized im2row implementation using INT8 arithmetic.

REFERENCES

Abtahi, T., Shea, C., Kulkarni, A., and Mohsenin, T. Accelerating


Barabasz, B. and Gregg, D. Winograd convolution for dnn: Beyond linear polynomials, 2019.


A APPENDIX

A.1 Winograd-aware layers for other architectures

The results of our study of Winograd-aware networks presented Section 6 showed multiple configurations of the ResNet-18 architecture at different width-multipliers, bit-widths, quantization levels and convolution algorithms. Here, we present a similar analysis for two other popular architectures for image classification. We limit our study to the full models (i.e. mult=1.0) We show results for SqueezeNet (Iandola et al., 2016) in Table 4 and for ResNeXt (Xie et al., 2017) in Table 5. These results align with what was observed for ResNet-18: In the presence of quantization, learning the Winograd transformations (flex configurations) resulted in superior performance than using the default (static) transformations. All experiments used the same hyper-parameters as described in Section 5.

<table>
<thead>
<tr>
<th>Conv. Type</th>
<th>Bits / act.</th>
<th>WA trans.</th>
<th>CIFAR-10</th>
<th>CIFAR-100</th>
</tr>
</thead>
<tbody>
<tr>
<td>im2row</td>
<td>-</td>
<td>91.13</td>
<td>69.06</td>
<td>74.54</td>
</tr>
<tr>
<td>WAf2</td>
<td>static</td>
<td>91.31</td>
<td>69.42</td>
<td></td>
</tr>
<tr>
<td>WAf2</td>
<td>flex</td>
<td>91.25</td>
<td>69.36</td>
<td></td>
</tr>
<tr>
<td>WAf4</td>
<td>static</td>
<td>91.23</td>
<td>69.14</td>
<td></td>
</tr>
<tr>
<td>WAf4</td>
<td>flex</td>
<td>91.41</td>
<td>69.32</td>
<td></td>
</tr>
</tbody>
</table>

Table 4. Comparison between standard convolutions (im2row) and Winograd-aware layers for SqueezeNet. With INT8 quantization and using the default transformation matrices (static), larger tile sizes (F4) introduce substantial numerical error and result in a severe accuracy drop. This drop in accuracy is significantly reduced if the transformations are learnt (flex).

Table 5. Comparison between standard convolutions (im2row) and Winograd-aware layers for ResNeXt(20(8×16)). With INT8 quantization and using the default transformation matrices (static), larger tile sizes (F4) introduce substantial numerical error and result in a severe accuracy drop. This drop in accuracy is significantly reduced if the transformations are learnt (flex).

For both architectures, INT8 Winograd-aware F4 models with learnt Winograd transformations did not result in a accuracy gaps as pronounced as the ones reported for ResNet-18 in Section 6. These models even surpass the im2row baselines for CIFAR-100. We argue this is because SqueezeNet and ResNeXt(8×16) have fewer 3×3 convolutional layers (8 and 6, respectively) compared to ResNet-18, which has 16. Therefore, the succession of fewer convolutional layers implemented as Winograd convolutions reduces the overall impact of numerical error.

A.2 Overhead of Learnt Winograd Transforms

The default Winograd transformation matrices contain varying amounts of 0’s. For F2 the sparsity ratios are 50%, 33% and 25% respectively for $B^T$, G and $A^T$. From the construction process of these matrices and specially the choice of polynomial points, we would expect lower sparsity ratios as these transforms are adjusted for larger input patches. For example, for the default transforms F1 these ratios are 22%, 22% and 25%. For implementations of matrix-matrix multiplications that can exploit data sparsity, as is the case of Arm’s Compute Library, having zeros means less compute which often translate into lower latencies.

The Winograd-aware formulation here presented doesn’t impose restrictions on how the learnt transform should look like. As a consequence, the resulting transforms rarely contain zeros. This translates in additional compute for input $B^T dB$ and output $A^T y.A$ transforms. The impact of using dense, learnt, transforms for WAf2 models running on a Cortex-A73 is a latency increase of 17% (+6ms) and 20% (+6ms) for FP32 and INT8 respectively for a ResNet18 network. This increase in latency is higher on the Cortex-A53 since the Winograd transforms are proportionally more expensive on this core. These penalties represent the worst case performance increase, assuming the transforms are compute bound. However, we believe that due to the access patterns of the Winograd transform kernels (gather and scatter across a wide area of memory) at least some of the performance of the transforms results from misses in the cache hierarchy and so some additional computation can be tolerated without necessarily increasing execution time.

We note that the impact for F2 models is considerably higher especially since the original transforms G and A are, not only sparse, but binary and the learnt ones are not. However, these penalties are never met in practice since F2 Winograd-aware models with default transforms can perform equally well as those with learnt transforms (as shown in Figure 4 and Tables 4 and 5) even in INT8.

Even with the performance loss due to the learnt transforms, we’re still demonstrating some (non-negligible) 1.54× and
1.43× speedup compared to INT8 im2row for A73 and A53 respectively. To the best of our knowledge this is the first time INT8 Winograd convolutions are empirically proven to work.

### A.3 Architectures optimized with wiNAS

Our framework wiNAS, takes a given macro-architecture and optimizes each 3 × 3 convolutional layer by choosing from direct convolution or different Winograd configurations. For the search, all 1 × 1 convolutions were fixed to use im2row.

For wiNASWA in FP32, the resulting architecture only substituted the last convolution layer with im2row instead of F2. The rest of the layers remained unchanged from the WA_F4 configuration (which was described in Section 5.1). The same micro-architecture was used in CIFAR-10 and CIFAR-100.

For wiNASWA with 8-bit quantization and CIFAR-10, wiNAS replaced the 5th and second last convolutional layers with im2row, instead of F4 and F2 respectively. For CIFAR-100, more optimization was compared to WA_F4. The resulting micro-architecture optimization is shown in Figure 9 (left).

When introducing quantization in the search space, wiNASWA-Q, the resulting architectures are shown in Figure 9 for both CIFAR-10 (middle) and CIFAR-100 (right).
Searching for Winograd-aware Quantized Networks

Figure 9. Resulting architectures after optimizing a ResNet-18 macro-architecture using wiNAS. For wiNAS\textsubscript{WA} and CIFAR-100, the architecture resulted is shown on the left. With wiNAS\textsubscript{WA-Q}, that introduces quantization in the search space, the optimization resulted in different architectures for CIFAR-10 (middle) and CIFAR-100 (right), evidencing the difference in complexity of the latter.