Accelerating Winograd Convolutions using Symbolic Computation and Meta-programming

Arya Mazaheri
Technical University of Darmstadt
Department of Computer Science
Germany
mazaheri@cs.tu-darmstadt.de

Tim Beringer
Technical University of Darmstadt
Department of Computer Science
Germany
tberinger@stud.tu-darmstadt.de

Matthew Moskewicz
Tesla Inc.
Palo Alto, CA, USA
mmoskewicz@tesla.com

Felix Wolf
Technical University of Darmstadt
Department of Computer Science
Germany
wolf@cs.tu-darmstadt.de

Ali Jannesari
Iowa State University
Department of Computer Science
Ames, IA, USA
jannesari@iastate.edu

Abstract

Convolution operations are essential constituents of convolutional neural networks. Their efficient and performance-portable implementation demands tremendous programming effort and fine-tuning. Winograd's minimal filtering algorithm is a well-known method to reduce the computational complexity of convolution operations. Unfortunately, existing implementations of this algorithm are either vendor-specific or hard-coded to support a small subset of convolutions, thus limiting their versatility and performance portability. In this paper, we propose a novel method to optimize Winograd convolutions based on symbolic computation. Taking advantage meta-programming and auto-tuning, we further introduce a system to automate the generation of efficient and portable Winograd convolution code for various GPUs. We show that our optimization technique can effectively exploit repetitive patterns, enabling us to reduce the number of arithmetic operations by up to 62% without compromising numerical stability. Moreover, we demonstrate in experiments that we can generate efficient kernels with runtimes close to deep-learning libraries, requiring only a minimum of programming effort, which confirms the performance portability of our approach.

Keywords Deep learning, Winograd convolution, meta-programming, symbolic computation

ACM Reference Format:
Arya Mazaheri, Tim Beringer, Matthew Moskewicz, Felix Wolf, and Ali Jannesari. 2020. Accelerating Winograd Convolutions using Symbolic Computation and Meta-programming. In *Fifteenth European Conference on Computer Systems (EuroSys '20), April 27–30, 2020, Heraklion, Greece.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3342195.3387549

1 Introduction

Convolutional neural networks (ConvNets) have emerged as the mainstream machine-learning method for a broad variety of computer-vision tasks, including object detection [9], image segmentation [19] and video classification [14]. Convolutional layers, particularly small ones with 3×3 filter sizes, are the main constituents of modern ConvNets, as they achieve higher accuracy with fewer parameters than shallow networks with larger filters [12, 29]. Such layers are used abundantly across the whole network and often dominate the computation and parameter amount. Therefore, speeding them up would have a great impact on alleviating the inference time and promoting the usage of ConvNets.

Direct convolution is a basic implementation with its computational requirements often exceeding available resources. Therefore, to further speed up the convolutional layers, new algorithmic improvements had to be introduced. Winograd's minimal filtering algorithms attempt to minimize the number of arithmetic operations for performing small convolutions [16]. The key idea is similar to the FFT-based convolution, where multiplication in the frequency domain corresponds to convolution in the time domain. FFT convolution transforms the input into the frequency domain using Discrete Fourier Transformation (DFT), multiplies by the frequency response of the filter, and then transforms it back into the time domain using the inverse DFT [22]. The Winograd convolution follows the same principle. Inputs and filters are first transformed into another space before the element-wise multiplication. After the multiplication step, the output will be transformed back to the original pixel space to obtain the final result. Unlike the FFT-based convolution, which uses complex numbers, all arithmetic operations of the Winograd convolution use real numbers, thus requiring fewer operations [16]. Lavin and Gray [16] showed that these algorithms could be around 2× faster than the direct convolution. However, due to the additional floating-point rounding errors, the results are not as accurate as the direct method.

Despite such a rewarding optimization, benefiting from Winograd's algorithm demands higher programming effort than the direct convolution to achieve ultimate performance. Inefficient data access patterns or unnecessary memory transfers may cost even more than the time saved by performing fewer computations. Consequently, performance engineers often develop multiple versions of the Winograd convolution, each supporting a different filter and output tile size. Such a rigid design prevents runtime frameworks from using Winograd convolutions more effectively for layers with different specifications, as the flexibility in choosing the output tile size is essential for achieving higher speedups. Within our experiments, we observed that inference frameworks and engines (e.g., cuDNN) often pick Winograd convolutions for a limited number of convolutional layers. Moreover, attaining the highest achievable performance across various platforms requires endless rounds of manual tuning to specialize the code to new hardware specifications.

In this paper, we introduce a new system in an attempt to address the issues mentioned above and generate a performance-portable Winograd convolution. From our point of view, we define *performance portability* as "to achieve a performance close to best-known vendor runtime on each platform with a single source code". To this end, we leverage symbolic computation to analyze Winograd transformations and identify repetitive terms ready for factorization. Along with additional optimization methods, we can generate minimal and efficient Winograd transformation code. This method dramatically reduces the arithmetic operations of Winograd transformation steps by up to 62%. Additionally, we use meta-programming and code generation to specialize the Winograd convolution code for each particular configuration and hardware platform. As a result, we do not need to manually tune or select optimizations according to each new target platform. We integrated our method in Boda [24], a ConvNet inference acceleration framework, and obtained competitive performance compared to other inference frameworks, such as cuDNN and MIOpen. Our experiments show that our method can even surpass the performance of these vendor-libraries across a subset of convolutions. In essence, this paper makes the following contributions:

- A novel method based on symbolic computation to generate efficient recipes for Winograd transformations
- A solution for selecting an appropriate output tile size to balance the numerical stability and efficiency of Winograd convolutions
- Generation of efficient Winograd algorithm code for any convolution operation using template meta-programming
- Performance portability via auto-tuning for Winograd convolutions on various GPUs, including mobile GPUs

In the remainder of the paper, we first explain the concept behind ConvNets and Winograd convolutions. Then, in Section 3 we discuss our approach, followed by evaluation results in Section 4. A concise review of related work is presented in Section 5. Finally, we conclude the paper in Section 6.

2 Background and Motivation

The essential idea behind ConvNets is their ability to learn a broad set of filters (a.k.a. kernels), organized into a hierarchy of layers, to extract meaningful information from a given image. Convolutional layers are the main constituents of ConvNets, which can be defined as the function $output = conv(input, filters)$, where $output$, $input$ and $filters$ are all multi-dimensional arrays (i.e. tensors). The main task is to cross-correlate a set of learned filters uniformly on various scopes of a given image using the sliding window approach. The output is a tensor called feature map, which contains abstract information, such as curves and edges. As we proceed from the first to the last layers of the network, we can observe the abstraction level of the features to rise. Therefore, deeper ConvNets are likely to perform better than shallow networks in various computer vision tasks.

The deeper a network becomes, the more parameters it usually includes. Hence, more data-dependent arithmetic operations will be involved, prolonging training and inference time. Furthermore, most of the computations take place in convolutional layers and accelerating their execution benefits the total inference time immensely. It is often possible to reshape the convolution operation as a matrix-multiplication operation. Thus, we can use highly efficient linear algebra libraries (BLAS), such as single-precision general matrix multiply (SGEMM). Such libraries are often highly parallelized and use GPUs for obtaining the highest speedups. Higher bandwidth, latency hiding via thread parallelism, and easily programmable registers make GPUs a lot faster than CPUs.

As the overall size of ConvNets grew, which is influenced by their depth, input size, and kernel size, the efficient execution of such networks gained more importance, leading to the introduction of inference frameworks such as TVM [4], TensorFlow's XLA [1], and Glow [28]. Such frameworks usually perform convolution operations as a series of dot-products, often with the assistance of a highly-efficient BLAS library implementation to maximize parallelism and runtime efficiency. They also perform various graph-level

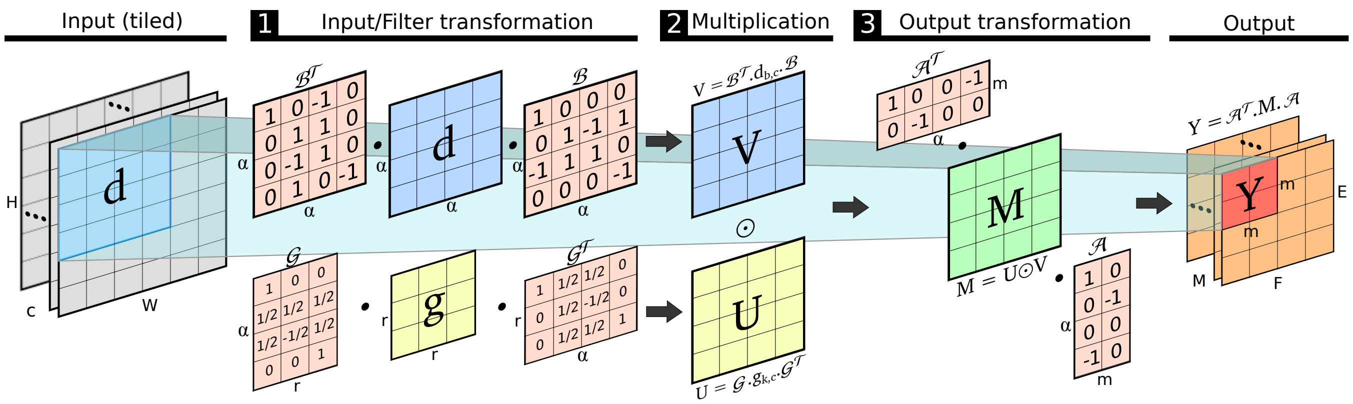


Figure 1. Visual representation of the computational steps within a sample $F(2^2, 3^2)$ Winograd convolution.

and code-level optimizations to minimize the memory footprint and accelerate the network inference time. Among the existing open-source frameworks, Boda [24] is mainly designed to accelerate ConvNet inference using template meta-programming to generate specialized code for various GPU platforms. Boda provides a flexible platform to write meta-code, from which low-level code optimized for given hardware can be generated. For instance, we can unroll the loops, generate a long sequence of memory instructions, and handle different input regimes.

2.1 Winograd convolution

A. Toom [30] and S. Cook [7] originally proposed optimal filtering algorithms using polynomial residuals. Afterward, Shmuel Winograd generalized these algorithms and proposed a method for the efficient computation of finite impulse response (FIR) filters [33]. Within this algorithm, computing m outputs with an r-tap FIR filter, which is denoted by $F(m, r)$ for a 1D convolution, requires $m + r - 1$ multiplications. Such a reduction is quite significant in comparison with the direct method, which requires $m \times r$ multiplications [16]. To explain how such a reduction can be achieved, we use $F(2, 3)$ as an example. For instance, for an input vector $d = (d_0, d_1, d_2, d_3)$ and $g = (g_0, g_1, g_2)$, the Winograd algorithm transforms the input data and the filter to $v = (v_0, v_1, v_2, v_3)$ and $u = (u_0, u_1, u_2, u_3)$, respectively, using the following equations:

$$v_0 = d_0 - d_2, \qquad u_0 = g_0 \tag{1}$$

$$v_1 = d_1 + d_2, \qquad u_1 = \frac{g_0 + g_1 + g_2}{2} \tag{2}$$

$$v_2 = d_2 - d_1, \qquad u_2 = \frac{g_0 - g_1 + g_2}{2} \tag{3}$$

$$v_3 = d_1 - d_3, \qquad u_3 = g_2 \tag{4}$$

Then, we multiply u and v element-wise and store the result in $c = u \odot v$, such that each element within c is denoted as $c_i = u_i \times v_i$. Lastly, the final result $y = (y_0, y_1)$ is computed using the following equation:

$$y_0 = c_0 + c_1 + c_2, \qquad y_1 = c_1 - c_2 - c_3 \tag{5}$$

The Winograd algorithm generalizes the transformations mentioned above and summarizes all these steps into a single equation, such that $y = \mathcal{A}[(\mathcal{G}g) \odot (\mathcal{B}d)]$, where the transformation matrices for $F(m, r)$ are:

$$\mathcal{A} = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 1 & -1 & -1 \end{bmatrix},$$

$$\mathcal{G} = \begin{bmatrix} 1 & 0 & 0 \\ 1/2 & 1/2 & 1/2 \\ 1/2 & -1/2 & 1/2 \\ 0 & 0 & 1 \end{bmatrix}, \mathcal{B} = \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix} \tag{6}$$

Thus, the Winograd algorithm consists of three main stages, of which the first and the last stages perform domain transformations. All transformations are done by matrix multiplications using pre-computed matrices ($\mathcal{A}$, $\mathcal{G}$, and $\mathcal{B}$) for each transformation. These matrices are fixed and are usually generated using the Toom-Cook method with a set of heuristically chosen polynomial points [2]. In a given $F(m^2, r^2)$ 2D Winograd convolution, filter size r and the output tile size m define the internal tile size (i.e., $\alpha = m + r - 1$), which in turn determines the shapes and values of the transformation matrices. Thus, each Winograd algorithm with distinct α demands a particular set of transformation matrices. Although the convolution operation determines the filter size r of the Winograd algorithm, the output tile size m can be freely chosen. Theoretically, by choosing a larger m, we can save more operations in the element-wise matrix multiplication step. However, it causes Winograd transformations to involve more elements, allowing floating-point rounding errors to jeopardize their numerical stability.

Figure 1 depicts all three stages of the computation for a sample Winograd convolution $F(2^2, 3^2)$. Before we explain each step, we define the following symbols, which are used by a Winograd convolution $F(m, r)$:

- $g_{k,c}$: k-th filter with channel c
- $d_{c,b}$: b-th input tile of c-th channel
- $Y_{k,b}$: b-th output tile for the k-th filter
- m: Winograd's output tile size
- r: kernel or filter size
- $P := N\lceil H/m \rceil \lceil W/m \rceil$: number of internal image tiles
- $\alpha := m + r - 1$: Winograd's internal working tile size
- $\mathcal{G}$, $\mathcal{B}$, and $\mathcal{A}$: transformation matrices for input, filter and output, respectively

2.1.1 Input and filter transformation

First, the input is decomposed into $\alpha \times \alpha$ tiles with the vertical and horizontal stride of $\alpha - r + 1$. This stride causes neighboring tiles to overlap by $r - 1$ elements. Each input tile and filter is then transformed by two transformation matrices $V = \mathcal{B}^T d_{c,b} \mathcal{B}$ and $U = \mathcal{G} g_{k,c} \mathcal{G}^T$.

2.1.2 Matrix multiplication

The main computation happens in this stage, where element-wise matrix multiplication is used for multiplying the transformed filter U with the transformed input of the same channel V. Then, all channels of the same image should be summed up ($M = \sum_{c=1}^{C} U_{k,c} V_{c,b}$).

2.1.3 Output transformation

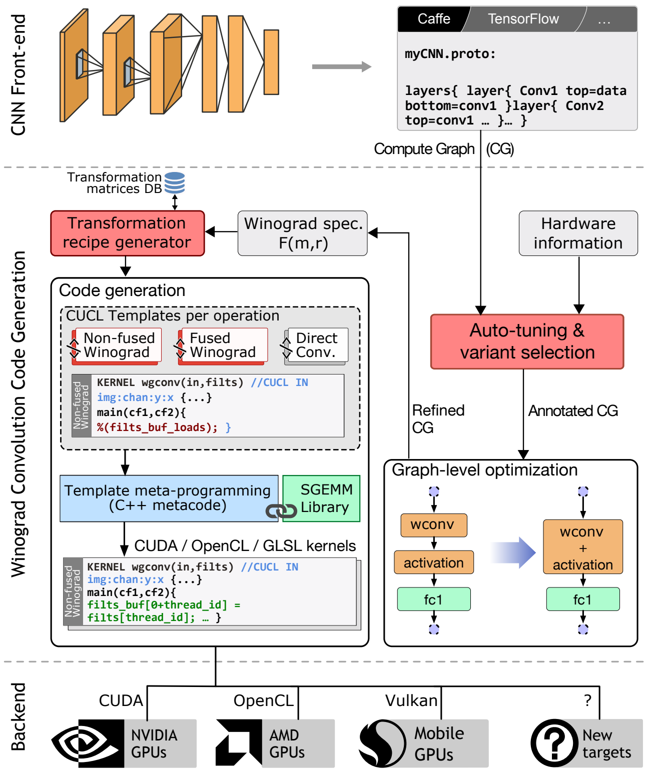
Finally, similar to the first stage, the output tiles are transformed back into the original space as $Y = \mathcal{A}^T M \mathcal{A}$. The $\alpha \times \alpha$ tiles are transformed into $m \times m$ tiles first, and then placed into the output image at their corresponding position.

2.2 Winograd convolution optimization

To fully benefit from the Winograd convolution, we need to pick a suitable output tile size m, which meets the expected accuracy level and memory limitations. Furthermore, various code optimizations (i.e., data layout and fast matrix multiplication) are usually needed to improve the overall runtime performance [13, 21]. However, such optimizations depend on Winograd specifications and the target hardware platform. Such a challenging task can be addressed effectively using an inference engine, capable of generating specialized code. Our proposed method follows this idea and is integrated into an inference framework.

3 Approach

A generic yet portable implementation invariably involves both low-level programming and a significant degree of meta-programming [5, 15]. Thus, we embrace both of them in an attempt to create a system for generating efficient and portable Winograd convolution code with any specification. We use the Boda framework as the basis for implementing our method. The input is a ConvNet model, which Boda parses it as a computational graph suitable for graph optimization and variant selection. Figure 2 depicts a high-level overview of our approach. Once the framework picks a Winograd convolution according to the hardware and the convolution parameters, we start with the specification of the selected convolution, denoted by $F(m, r)$. First, we generate corresponding transformation matrices, after which we use Winograd templates to generate efficient code. Depending on the desired GPU platform, we can generate CUDA, OpenCL, or GLSL code. The resulting GPU kernels are compiled alongside their host code into a binary file, which can be executed in a standalone fashion on the target device. Below, we explain the main components of our method in more detail.



```
myCNN.proto:

layers{ layer{ Conv1 top=data
bottom=conv1 }layer{ Conv2
top=conv1 … }… }
```

```
KERNEL wgconv(in,filts) //CUCL IN
img:chan:y:x {...}
main(cf1,cf2){
%(filts_buf_loads); }
```

```
KERNEL wgconv(in,filts) //CUCL IN
img:chan:y:x {...}
main(cf1,cf2){
filts_buf[0+thread_id] =
filts[thread_id]; … }
```

Figure 2. The workflow of generating performance portable Winograd convolution code.

3.1 Winograd transformation optimization

Small convolutions are the primary beneficiaries of the Winograd algorithm. Moving toward larger filters or output tiles makes the Winograd algorithm prone to low precision and low performance. The accuracy degrades after a multitude of data-scaling and floating-point operations with finite precision [2, 32]. The performance also deteriorates because larger Winograd convolutions demand larger transformation matrices. Existing Winograd implementations attempt to perform a multitude of matrix multiplications in order to transform input tiles, filters, and outputs into the desired domain. The growing number of arithmetic operations involved in the transformation steps gradually becomes a burden. In this section, we provide a solution to optimize these steps and alleviate the adverse effects of the Winograd transformations.

3.1.1 Selecting polynomial points

A critical factor in improving the numerical accuracy of Winograd convolutions is to find a good set of polynomial points as the basis for generating transformation matrices. Inspired by B. Barabasz et al. [2], our method heuristically finds the polynomial points and uses the modified Toom-Cook method to generate transformation matrices—based on the idea of evaluating polynomials at given points using the Lagrange interpolation theorem [2].

For a Winograd convolution $F(m^2, r^2)$, we need $m + r - 2$ points. We begin with the ordered set $(0, -1, 1)$, which has been proven to provide ideal points for reducing arithmetic operations and maintaining high accuracy because multiplication by 1 or -1 can simply be skipped, and multiplication by zero enables us to skip both the scaling and addition [2]. When more than three points are required, we perform an exhaustive search for the remaining points. Empirical evaluations show that small and simple integers and fractions are good candidates for reducing the required number of scalings and additions [2]. We follow the same approach and select the points P as rational numbers, where $P = \{\frac{a}{b} | a, b \in \mathbb{Z}, -9 \leqslant a \leqslant 9, 1 \leqslant b \leqslant 9\}$. To find the most accurate set of points, we iteratively examine the precision of Winograd convolution results. In each iteration, we create random input and filter tensors with a uniform distribution in the range of (-1, 1) because, in practice, the weights of deep neural networks are primarily concentrated in this range. To obtain the highest precision, we compare the results (FP32) with direct convolution (FP64) and compute the error rate using the L1 norm. We perform this analysis 10,000 times, a relatively high iteration count to make the results stable. We select the median value as the representative error rate of the chosen points.

3.1.2 Transformation recipe generation

Transformation matrices for a given Winograd convolution are always the same and often follow a regular pattern. Thus, we avoid running an ordinary matrix-multiplication kernel and, instead, we use symbolic computation to intensively simplify the transformation steps into a sequence of instructions for constructing the transformed matrices. First, we use the modified Toom-Cook method to generate the transformation matrices $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{G}$ based on the selected polynomial points. One crucial feature is that we use rational numbers instead of real floating-point numbers to avoid rounding errors. Then, we create a symbol matrix with its size equal to the input size, similar to:

$$\mathcal{G} = \begin{bmatrix} -1 & 0 & 0 \\ 1/2 & 1/2 & 1/2 \\ 1/2 & -1/2 & 1/2 \\ 0 & 0 & 1 \end{bmatrix}, g = \begin{bmatrix} g_{0,0} & g_{0,1} & g_{0,2} \\ g_{1,0} & g_{1,1} & g_{1,2} \\ g_{2,0} & g_{2,1} & g_{2,2} \end{bmatrix} \quad (7)$$

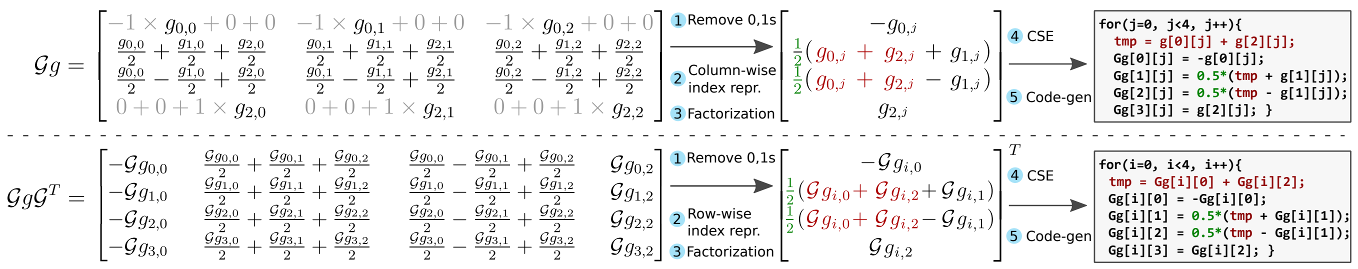
We multiply the transformation matrix with the symbol matrix and obtain the result. Next, we apply the following sequence of steps to optimize the transformed matrices:

1. **Elimination of unnecessary arithmetic operations:** We observe many multiplications by one or zero, and additions with zero (i.e. $1 \times g_{i,j} + 0$ or $0 \times g_{i,j}$), which we eliminate and simplify down to $g_{i,j}$ or 0.
2. **Column-/row-wise index-based representation:** We transform the resulting matrix into a vector with the variable subscripts replaced by an induction variable symbol. A 2D Winograd transformation consists of two consecutive matrix multiplications. We realized that we can always apply a column-wise and row-wise representation generalization to these multiplications, respectively. The ultimate goal is to generate the transformed matrix using only a single loop construct. Additionally, we can unroll the loops if necessary.
3. **Factorization:** Each row within the resultant vector contains terms with rational coefficients. In the case we find common coefficients across the terms, we apply factorization to save redundant multiplications.
4. **Common sub-expression (CSE) elimination:** We use the CSE algorithm to find the common terms among the vector rows. Thus, we can compute them once and reuse them multiple times. This method reduces both the number of additions and multiplications.

Figure 3 illustrates the above steps applied to the sample filter defined in Equation 7. These optimizations need to be performed only once before the actual Winograd convolution execution to obtain the transformation recipes. Since these recipes remain the same for every specific $F(m, r)$, we store them in a database to facilitate their reuse and avoid generating them again.

3.2 Code generation

We use CUDA and the rather new Vulkan API to target the GPU platforms within our study. We decided to use Vulkan instead of OpenCL on non-Nvidia GPUs, as it supports a broader range of GPUs, including mobile platforms. Furthermore, evidence shows that the Vulkan compiler can produce more optimized GPU codes compared with OpenCL compilers [23]. However, CUDA and Vulkan programming interfaces are considerably different. Thus, generating a GPU kernel out of a single code template might seem implausible at first sight. Nevertheless, in our previous work, we extended the Boda framework by adding a high-level GPU



$$\mathcal{G}g = \begin{bmatrix} -1\times g_{0,0}+0+0 & -1\times g_{0,1}+0+0 & -1\times g_{0,2}+0+0 \\ \frac{g_{0,0}}{2}+\frac{g_{1,0}}{2}+\frac{g_{2,0}}{2} & \frac{g_{0,1}}{2}+\frac{g_{1,1}}{2}+\frac{g_{2,1}}{2} & \frac{g_{0,2}}{2}+\frac{g_{1,2}}{2}+\frac{g_{2,2}}{2} \\ \frac{g_{0,0}}{2}-\frac{g_{1,0}}{2}+\frac{g_{2,0}}{2} & \frac{g_{0,1}}{2}-\frac{g_{1,1}}{2}+\frac{g_{2,1}}{2} & \frac{g_{0,2}}{2}-\frac{g_{1,2}}{2}+\frac{g_{2,2}}{2} \\ 0+0+1\times g_{2,0} & 0+0+1\times g_{2,1} & 0+0+1\times g_{2,2} \end{bmatrix} \longrightarrow \begin{bmatrix} -g_{0,j} \\ \frac{1}{2}(g_{0,j}+g_{2,j}+g_{1,j}) \\ \frac{1}{2}(g_{0,j}+g_{2,j}-g_{1,j}) \\ g_{2,j} \end{bmatrix} \longrightarrow$$

```
for(j=0, j<4, j++){
  tmp = g[0][j] + g[2][j];
  Gg[0][j] = -g[0][j];
  Gg[1][j] = 0.5*(tmp + g[1][j]);
  Gg[2][j] = 0.5*(tmp - g[1][j]);
  Gg[3][j] = g[2][j]; }
```

$$\mathcal{G}g\mathcal{G}^T = \begin{bmatrix} -\mathcal{G}g_{0,0} & \frac{\mathcal{G}g_{0,0}}{2}+\frac{\mathcal{G}g_{0,1}}{2}+\frac{\mathcal{G}g_{0,2}}{2} & \frac{\mathcal{G}g_{0,0}}{2}-\frac{\mathcal{G}g_{0,1}}{2}+\frac{\mathcal{G}g_{0,2}}{2} & \mathcal{G}g_{0,2} \\ -\mathcal{G}g_{1,0} & \frac{\mathcal{G}g_{1,0}}{2}+\frac{\mathcal{G}g_{1,1}}{2}+\frac{\mathcal{G}g_{1,2}}{2} & \frac{\mathcal{G}g_{1,0}}{2}-\frac{\mathcal{G}g_{1,1}}{2}+\frac{\mathcal{G}g_{1,2}}{2} & \mathcal{G}g_{1,2} \\ -\mathcal{G}g_{2,0} & \frac{\mathcal{G}g_{2,0}}{2}+\frac{\mathcal{G}g_{2,1}}{2}+\frac{\mathcal{G}g_{2,2}}{2} & \frac{\mathcal{G}g_{2,0}}{2}-\frac{\mathcal{G}g_{2,1}}{2}+\frac{\mathcal{G}g_{2,2}}{2} & \mathcal{G}g_{2,2} \\ -\mathcal{G}g_{3,0} & \frac{\mathcal{G}g_{3,0}}{2}+\frac{\mathcal{G}g_{3,1}}{2}+\frac{\mathcal{G}g_{3,2}}{2} & \frac{\mathcal{G}g_{3,0}}{2}-\frac{\mathcal{G}g_{3,1}}{2}+\frac{\mathcal{G}g_{3,2}}{2} & \mathcal{G}g_{3,2} \end{bmatrix} \longrightarrow \begin{bmatrix} -\mathcal{G}g_{i,0} \\ \frac{1}{2}(\mathcal{G}g_{i,0}+\mathcal{G}g_{i,2}+\mathcal{G}g_{i,1}) \\ \frac{1}{2}(\mathcal{G}g_{i,0}+\mathcal{G}g_{i,2}-\mathcal{G}g_{i,1}) \\ \mathcal{G}g_{i,2} \end{bmatrix}^T \longrightarrow$$

```
for(i=0, i<4, i++){
 tmp = Gg[i][0] + Gg[i][2];
 Gg[i][0] = -Gg[i][0];
 Gg[i][1] = 0.5*(tmp + Gg[i][1]);
 Gg[i][2] = 0.5*(tmp - Gg[i][1]);
 Gg[i][3] = Gg[i][2]; }
```

Figure 3. Illustration of a Winograd filter transformation being optimized prior to code generation.

interface capable of bridging syntactic incompatibilities [23]. This interface allows a specialized GPU kernel to be generated without modifying the main source code for every platform and Winograd specification.

At a high level, we choose to take a general and flexible approach to meta-programming. Rather than using language-level meta-programming, we write code generators directly in C++. We use Boda's native support for tensors at the meta-code layer to allow code generation to exploit fixed, exact sizes for all inputs and outputs. We observed that platform-specific compilers often do not successfully unroll loops and remove unneeded conditionals. In such cases, we directly emit a sequence of instructions for iterating through tensor elements, loading and storing data from/to global memory, shared memory, and registers. To do this, we move the loop to the meta-code level and replace it entirely with a template placeholder, such as `%(filts_buf_loads)`, `%(winograd_filt_transform)`, and `%(store_results)`. Then, at the meta-code level, we write code to generate the required sequence of instructions.

In general, we aim to make the code as simple as possible by reducing the usage of loops and conditions. Such an optimization increases the chance that the platform-specific compiler generates efficient binary code. Furthermore, the code generator has the privilege to employ the highly-tuned BLAS libraries that exist on the target platform, such as cuBLAS and CLBLast [25]. In this study, we used CLBlast to perform the matrix multiplications.

3.2.1 Winograd transformation meta-code

In a 2D Winograd convolution, each transformation step should perform two consecutive matrix multiplications to transform a given tensor into the desired domain. Such multiplications are costly, and depending on the matrix dimensions, they might impose significant runtime overhead. We aim to replace the six matrix multiplications (i.e. $\mathcal{G}.g.\mathcal{G}^T$, $\mathcal{B}^T.d.\mathcal{B}$, and $\mathcal{A}^T.M.\mathcal{A}$) with their corresponding intensively simplified single-level loops. We replace each matrix multiplication code with a template placeholder, such as `%(winograd_filt_transform)`, and later use meta-programming to fill in the placeholder with the transformation recipes that we introduced in Section 3.1. An example of the resulting code is illustrated in Figure 3. We further reduce the complexity and improve the performance of the transformation code using two optimization techniques:

- **Adaptive loop-unrolling:** We unroll the Winograd transformation loops to eliminate control instructions and achieve higher speedups. The unrolling factor is a tunable parameter, which we can tune according to available instruction cache size. For those loops in which the iteration count is not dividable by the unrolling factor, we find the closest divisor, or if we cannot find one, we fully unroll the loop.
- **Fused multiply-add (FMA) operations:** Often, the terms involved in computing the elements of transformed matrices contain a multiplication and an addition. Therefore, we can convert those operations into an FMA instruction and perform them all in one step, with a single rounding. We can benefit from FMA operations, provided that the target GPU and the programming interface support such operations. Otherwise, we avoid calling these instructions and simply rely on basic arithmetic instructions, instead.

3.2.2 Winograd code templates

We can implement Winograd convolutions in two different ways: (1) non-fused and (2) fused. A fused implementation is often preferable as it reduces the GPU memory transfer by merging all the Winograd steps into a single kernel. However, for large kernels, its memory requirement might exceed the available GPU shared memory space. In such cases, we use the non-fused implementation as a fallback. As a rule of thumb, fused implementations are often better suited for small convolutions such as 3×3 convolutions with small output tile sizes. We implemented code templates for both versions to demonstrate the applicability of each version under different circumstances.

Non-fused implementation

This version implies that we have a separate kernel for each step of the Winograd algorithm, and each kernel has to write the results back to the global memory. Although it increases the data transfer overhead, the non-fused version is still a viable option for larger Winograd convolutions, where the available shared memory is limited, particularly on mobile GPUs. In general, each kernel assigns a tile of data to each thread, which first loads the data from the global memory to its registers. Then, the actual computation takes place, and ultimately, the output will be written back into the global memory.

Despite the additional arithmetic operations caused by Winograd transformations, a significant portion of the computation takes place in Winograd's matrix multiplication step. Therefore, obtaining higher efficiency also relies on using optimized BLAS routines. The element-wise multiplication of the transformed input with transformed filters can be seen as a dot product of two vectors U and V, as we need to sum up the results across the channels. Therefore, we follow Lavin and Gray's [16] approach to pose this problem as a typical matrix multiplication and benefit from highly-efficient SGEMM routines.

To reframe the problem as an SGEMM operation, we vertically stack each element of a transformed filter tile. Then, we group them by their index within the tile and sort them by their filter K in ascending order, such that each group denotes the K-th elements of each filter tile. Consequently, all channels will be horizontally stacked and placed in their corresponding row, such that the first row contains all channels of the first element within the first filter tile. The resulting $(\alpha^2 K, C)$ matrix is $U'_{ij} = U^{i/K}_{i\%K,j}$. The transformed image is reframed in the opposite way, where elements with the same index are horizontally stacked and grouped by the same image tile. Then, the channels are stacked in a column-wise fashion.

Since both matrices U' and V' are grouped by the index within their tile, there will be α^2 groups per matrix. Only the groups in U' and V' sharing the same tile index have to be multiplied, and only α^2 matrix multiplications are needed. Since we need to multiply several small matrices, we avoid invoking different matrix multiplication kernels and, instead, use a batched-SGEMM operation to perform all the multiplications. We use a vendor BLAS library if it exists on the target platform. Otherwise, we use a self-developed SGEMM kernel in the Boda framework.

Fused implementation

Merging all the Winograd steps into a single kernel has the potential advantage of improving the usage of shared memory and registers. Data resides in the shared memory as long as it is needed for computation. Previously, Lavin and Gray [16] proposed this optimization solely for small Winograd configurations, such as $F(3,2)$ and $F(3,4)$, since the shared memory space is very limited [16]. We further extended this optimization and used meta-programming to support larger configurations, as long as enough shared memory is available. We split the threads within a thread block in half, such that the first half computes filter transformations, and the other half computes input transformations. The matrix multiplication step is divided into equal parts and distributed among the threads. Therefore, we cannot invoke a real matrix-multiplication kernel and, instead, implemented the multiplications within the kernel. Finally, all threads perform the output transformation together.

Table 1. Tuning parameters for Winograd convolutions.

Tuning Parameter	Purpose	Values
WV	Winograd variant (fused / non-fused)	[0, 1]
LU	Loop unrolling factor	[1, 2, 4, 6, ∞]
MNt	SGEMM Register blocking size	Exponential of two
MNb	SGEMM Thread blocking size	Exponential of two
m	Winograd output tile size	$2 \leq m \leq 10$

3.3 Variant selection and auto-tuning

It is generally hard, even with meta-programming, to develop comprehensive yet efficient Winograd convolution kernels that can run across various inputs and hardware platforms. We have two different variants of Winograd convolutions (i.e., fused and non-fused), which might perform differently on each target platform. Thus, based on the convolution and available resources, we expect to run the best performing variant on a given platform. Furthermore, each variant needs to be optimized prior to execution using several tuning parameters, as described in Table 1. By performing a brute-force, guided, or sampled exploration of the space of variants and tuning parameters, we can find the best parameters for a given Winograd convolution operation and provide performance portability among different hardware platforms. Considering the manageable size of the search space, we used the brute-force method. Nevertheless, the tuning process could be further accelerated using more sophisticated search methods.

4 Experimental Results

In this section, we assess different aspects of the proposed method by answering the following specific questions:

- How precise are the Winograd convolution results, and how does the accuracy change for different Winograd configurations?
- To what extent can we optimize the Winograd-transformation code?
- What is the runtime performance of the proposed method, and how does it perform compared with other deep-learning libraries?
- How portable are the generated Winograd convolutions across different GPU platforms?
- Is there any way to pick the most suitable configuration for a given Winograd convolution?

Table 2. Experimental setup.

	Nvidia GTX 1080Ti	AMD RX 580	ARM Mali G71
OS	Ubuntu 16.04 64-bit		Android 7.0
CPU	Intel Xeon Gold 6126, 12Core @ 2.6GHz		Cortex A73,A53
Host Memory	64 GB		3GB
GPU Memory	11GB GDDR5X	8GB GDDR5	-
Driver	Linux Display 410.66	AMDGPU-PRO 17.40	Native driver
Libraries	CUDA 10, cuDNN 7.3	MIOpen 2.1	ARM Compute Library v20.02.1

Experimental setup

To evaluate the proposed method, we chose NVIDIA GTX 1080 Ti and AMD Radeon RX 580, two popular desktop GPUs. We also used a mobile platform based on the Hikey 960 development kit, which contains an ARM Mali-G71 MP8 GPU. Table 2 summarizes the configuration details of the target platforms.

4.1 Accuracy analysis

As mentioned earlier in Section 3, moving toward a larger internal tile size α, which itself depends on the output tile size m and the filter size r, leads to a higher accuracy loss. However, it is not apparent how much error is tolerable during the inference phase. Thus, we measured the accuracy of Winograd convolutions with various internal tile sizes to find out how significant their error rates are and which one is probably more suitable for a convolutional layer.

Table 3 reports the selected polynomial points for different Winograd internal tile sizes, within the range of $[4, 16]$ alongside their relative error. We compute the relative error using the L1 norm $||X||_1$:

$$RelativeError = \frac{||\widehat{X} - X||_1}{||X||_1}$$
$$||X||_1 = max_j \sum_i |a_{i,j}|$$

, where $\widehat{X}$ and X are the Winograd (32-bits) and direct convolution (64-bits) results, respectively. Conventionally, the previous points can be reused for adding a new point to the sequence [2]. However, we noticed that by recomputing the whole sequence of points, more accurate results could be obtained.

We investigated the numerical stability of the generated Winograd convolutions by measuring their error range and error growth rate. Figure 4 depicts a box plot of L1-norm errors for Winograd convolutions with different α. We ran each case 10,000 times with randomly generated input and filters between $(-1, 1)$. We observed that the error rates proliferate with the addition of each new polynomial point. However, it does not precisely follow an exponential trajectory, as opposed to the observation made by Barabasz et al. [2]. Instead,

Table 3. Polynomial points selected by our method alongside their relative error.

α	Points	Relative Error
4	$BP = (0, 1, -1)$	6.11×10^{-8}
5	$BP \cup (2)$	2.65×10^{-7}
6	$BP \cup (1/2, -2)$	5.59×10^{-7}
7	$BP \cup (1/2, -2, 2)$	1.14×10^{-6}
8	$BP \cup (2, -1/2, 1/2, -2)$	1.76×10^{-6}
9	$BP \cup (2, -1/2, 1/2, -2, 4)$	9.93×10^{-6}
10	$BP \cup (1/2, -2, 2, -1/2, 4/3, -3/4)$	1.42×10^{-5}
11	$BP \cup (1/2, -2, 2, -1/2, 4/3, -3/4, -4)$	8.38×10^{-5}
12	$BP \cup (1/2, -2, 2, -1/2, 3/4, -4/3, 9/2, -2/9)$	1.83×10^{-4}
13	$BP \cup (1/2, -2, 2, -1/2, 4/3, -3/4, 1/4, -4, 4)$	5.36×10^{-4}
14	$BP \cup (1/2, -2, 2, -1/2, 9/7, -7/9, 1/4, -4, 7/9, -7/9)$	9.10×10^{-4}
15	$BP \cup (1/2, -2, 2, -1/2, 4/3, -3/4, 1/4, -4, 7/9, -9/7, 4)$	3.45×10^{-3}
16	$BP \cup (1/2, -2, 2, -1/2, 4/3, -3/4, 2/7, -7/2, 4/5, -5/4, 4, -1/4)$	4.66×10^{-3}

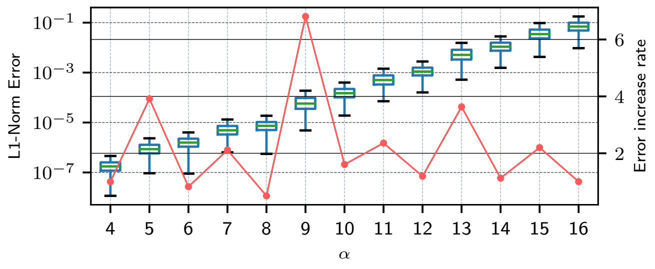


Figure 4. L1-norm error analysis for various Winograd internal tile sizes.

we noticed that Winograd convolutions with even α benefit from a lower error-growth rate. We observed the lowest error growth when $\alpha = 8$.

In comparison with the inference phase, accuracy is a more crucial factor for the training phase, as it affects learning stability. Nonetheless, according to previous studies [8, 10], error rates lower than 1e−02 do not harm the stability, implying that the inference phase is immune to such error rates. Such an observation suggests that our generated Winograd convolutions can be used during inference without experiencing any instability.

4.2 Winograd transformation optimization results

In Section 3.1, we proposed a method for reducing the computational complexity of Winograd transformation steps using symbolic computation. To demonstrate the effectiveness of our method, we numerically analyze the computations involved in Winograd transformation steps by directly counting the number of additions and multiplications. We selected Winograd convolutions with $m \in \{m \in \mathbb{N} | 2 \leq m \leq 10\}$ and $r \in \{3, 5, 7\}$. The results are given in Figures 5a– 5c. For each step, we separated the results into three columns, each representing a particular filter size. The baseline is the straightforward implementation of Winograd transformations using typical matrix multiplications. The optimized version represents the actual number of arithmetic operations involved in the generated code. We also demonstrate the amount of FMA operations, which we were able to identify.

We observed that our method was able to reduce the number of arithmetic operations in transformation steps by up to 62%. We annotated the highest amount of reductions in each diagram. We often obtain the highest amount of reduction when $\alpha = 8$, except for a few cases, where other internal tile sizes yield higher reductions. However, when looking at all the transformation steps, as depicted in Figure 5d, we can conclude that transformations are better suited for optimization when $\alpha = 8$. Such a value for α enables our method to factorize more common terms and improve data reuse. Since transformation steps might be considered as a small portion of the total computations involved in Winograd convolution, we also show the total amount of arithmetic reduction in Figure 5d. As the blue line indicates, the total reduction of arithmetic operations can reach up to 40%. Overall, our analysis of both accuracy and the number of arithmetic operations in Winograd transformations after optimization confirms that when $\alpha = 8$, Winograd convolutions can be optimized to a greater extent.

To further validate the effectiveness of our method, we generated CUDA kernels for sample convolutions with $r \in \{3, 5, 7\}$ and the same set of Winograd output tile sizes $m \in \{m \in \mathbb{N} | 2 \leq m \leq 10\}$. Figure 6 depicts all the runtimes for the convolutions that we ran on our Nvidia GPU. Our results suggest that Winograd convolutions with a filter size larger than five are probably not suitable for deployment, as other types of convolutions perform much faster. We further noticed that larger values of m do not necessarily save more operations during the matrix-multiplication step, as they cause additional computation overhead. Evidence [13] suggests that this happens mainly for two reasons: (1) The dimension of output images has to be divisible by m. Otherwise, the image is zero-padded, leading to a higher amount of operations during both transformation and matrix multiplication. (2) The amount of operations for the image and filter transformations grows quadratically with m.

For 3×3 convolutions with small batch sizes, smaller Winograd output tile sizes m offer better runtime. However, when we increase the batch size, larger values of m between (5,7) leads to a better result. In contrast, we observe a different behavior with 5×5 convolutions. For almost every batch size, an output tile size of $m = 4$ offers lower runtime, provided that we enable our optimization. We notice that our method can speed up convolutions by up to 1.65×, particularly when $\alpha = 8$.

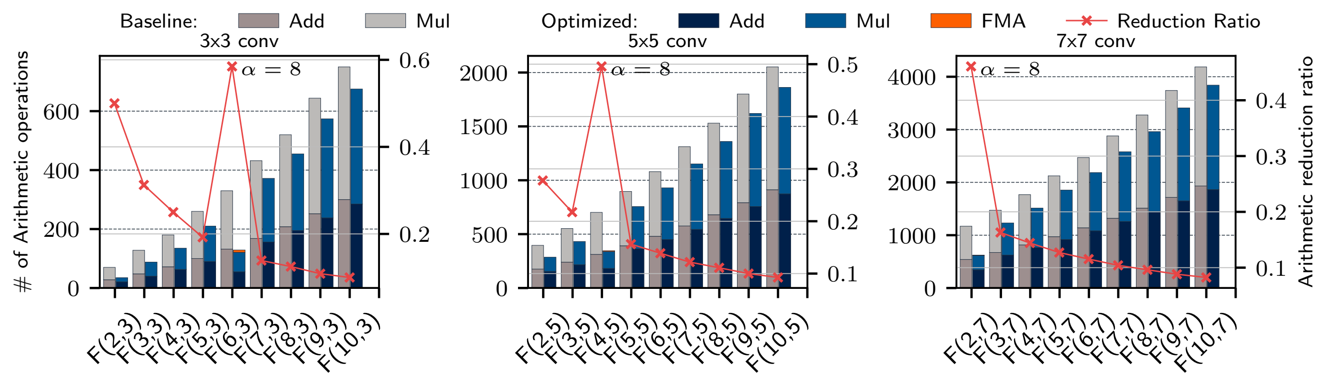
4.3 Efficiency and performance portability

To evaluate the runtime performance and performance portability of our approach, we selected a range of convolution operations and generated the corresponding GPU kernels for various platforms. To get the best performance, we used CUDA for Nvidia GPUs and Vulkan for AMD and Mali-G71 mobile GPUs.

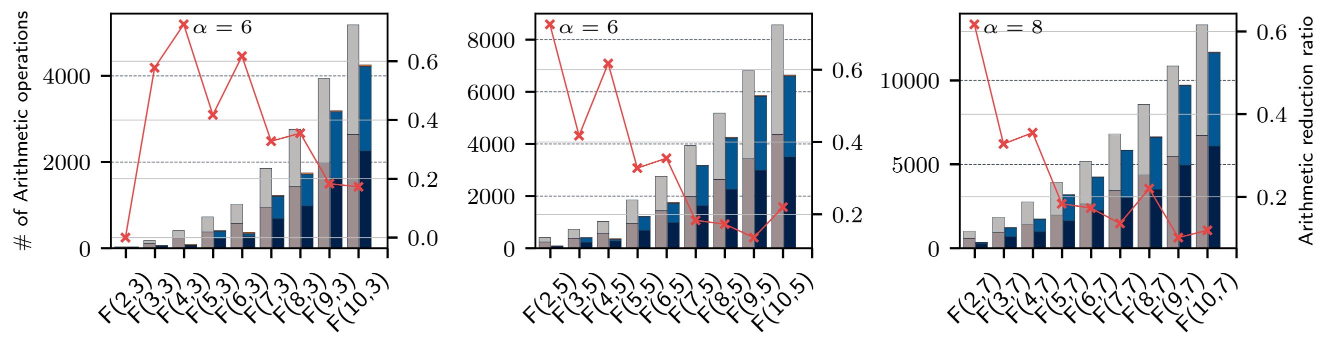
Instead of showing end-to-end runtime results for whole deep neural networks, we follow a more fine-grained approach and report per-convolution runtimes because it better highlights the impact of our optimizations. Successfully speeding up even a single convolutional layer implies shorter inference runtime for the whole network. Thus, we extracted 31 unique convolutions from AlexNet, Network-in-Network, and the InceptionV1 networks, which have (1) batch sizes of one and five, and (2) more than $1e8$ FLOPS. The rationale behind this selection is that we wanted these convolutions to model both a single inference and a streaming deployment scenario with a high computational load but some latency tolerance. The exact specifications for each of these 31 convolutions can be found in Table 4. For the sake of precision, we measured the execution times using GPU timers. Furthermore, to counter run-to-run variation, we executed each kernel ten times and reported the average of the runtimes we obtained. All the average speedups reported across the convolutions are computed using the geometric mean.

Table 4. KSZ, S, P, OC, and B are the kernel size, stride, padding, number of output channels, and batch size of each convolution operation. *in* and *out* are the sizes of input and output, specified as $y \times x \times chan$; FLOPs is the per-operation FLOP count.

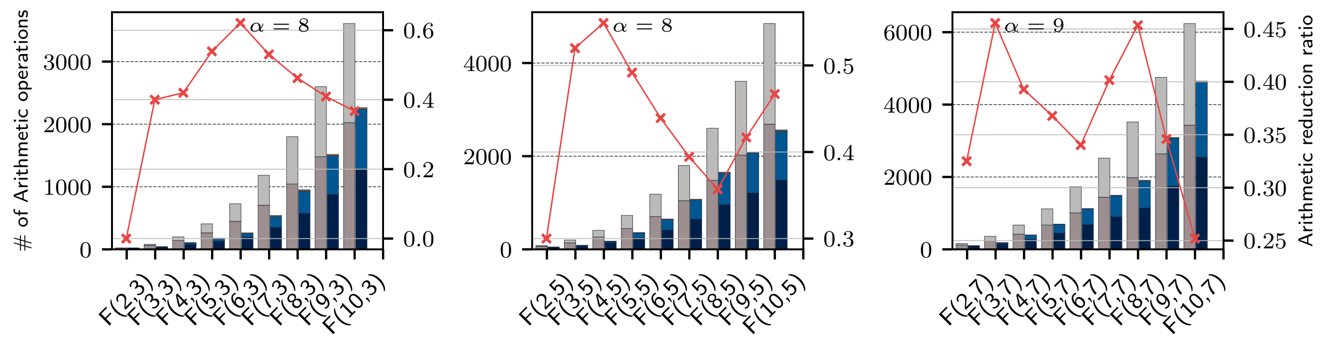
FLOPs	KSZ	S	P	OC	B	in
1e+08	5	1	2	32	5	$28 \times 28 \times 16$
1e+08	5	1	2	64	5	$14 \times 14 \times 32$
1.16e+08	3	1	1	256	1	$14 \times 14 \times 128$
1.2e+08	5	1	2	96	1	$28 \times 28 \times 32$
1.46e+08	3	1	1	288	1	$14 \times 14 \times 144$
1.73e+08	3	1	1	128	1	$28 \times 28 \times 96$
1.81e+08	3	1	1	320	1	$14 \times 14 \times 160$
2.01e+08	5	1	2	128	5	$14 \times 14 \times 32$
2.26e+08	3	1	1	320	5	$7 \times 7 \times 160$
2.55e+08	3	1	1	1024	1	$6 \times 6 \times 384$
2.99e+08	3	1	1	256	1	$13 \times 13 \times 384$
2.99e+08	3	1	1	384	1	$13 \times 13 \times 256$
3.25e+08	3	1	1	384	5	$7 \times 7 \times 192$
3.47e+08	3	1	1	192	1	$28 \times 28 \times 128$
3.52e+08	3	1	1	208	5	$14 \times 14 \times 96$
4.43e+08	3	1	1	224	5	$14 \times 14 \times 112$
4.49e+08	3	1	1	384	1	$13 \times 13 \times 384$
5.78e+08	3	1	1	256	5	$14 \times 14 \times 128$
6.02e+08	5	1	2	96	5	$28 \times 28 \times 32$
6.94e+08	3	1	1	192	1	$56 \times 56 \times 64$
7.32e+08	3	1	1	288	5	$14 \times 14 \times 144$
8.67e+08	3	1	1	128	5	$28 \times 28 \times 96$
8.96e+08	5	1	2	256	1	$27 \times 27 \times 96$
9.03e+08	3	1	1	320	5	$14 \times 14 \times 160$
1.27e+09	3	1	1	1024	5	$6 \times 6 \times 384$
1.5e+09	3	1	1	384	5	$13 \times 13 \times 256$
1.5e+09	3	1	1	256	5	$13 \times 13 \times 384$
1.73e+09	3	1	1	192	5	$28 \times 28 \times 128$
2.24e+09	3	1	1	384	5	$13 \times 13 \times 384$
3.47e+09	3	1	1	192	5	$56 \times 56 \times 64$
4.48e+09	5	1	2	256	5	$27 \times 27 \times 96$



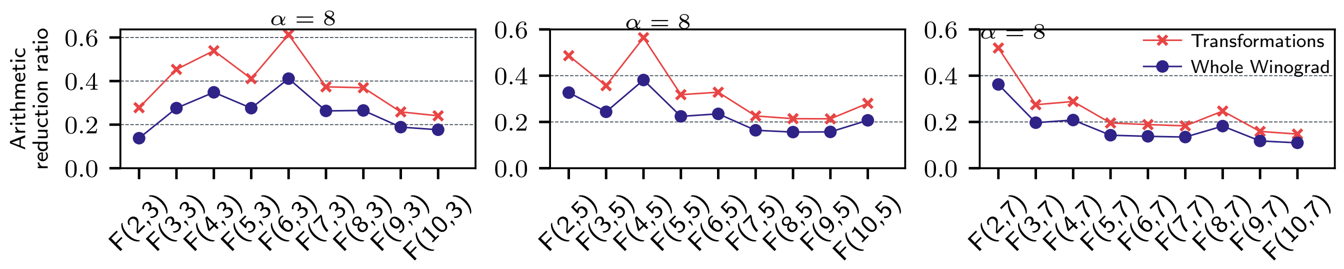
(a) Filter transformation



(b) Input transformation



(c) Output transformation



(d) Overall arithmetic reduction ratios related to transformation steps and the whole Winograd algorithm for a single tile.

Figure 5. Comparing the number of arithmetic operations of each Winograd transformation, before and after optimization, where $r \in \{3, 5, 7\}, m \in [2, 10]$. Our analysis indicates that the highest arithmetic reduction can be achieved when $\alpha = 8$.

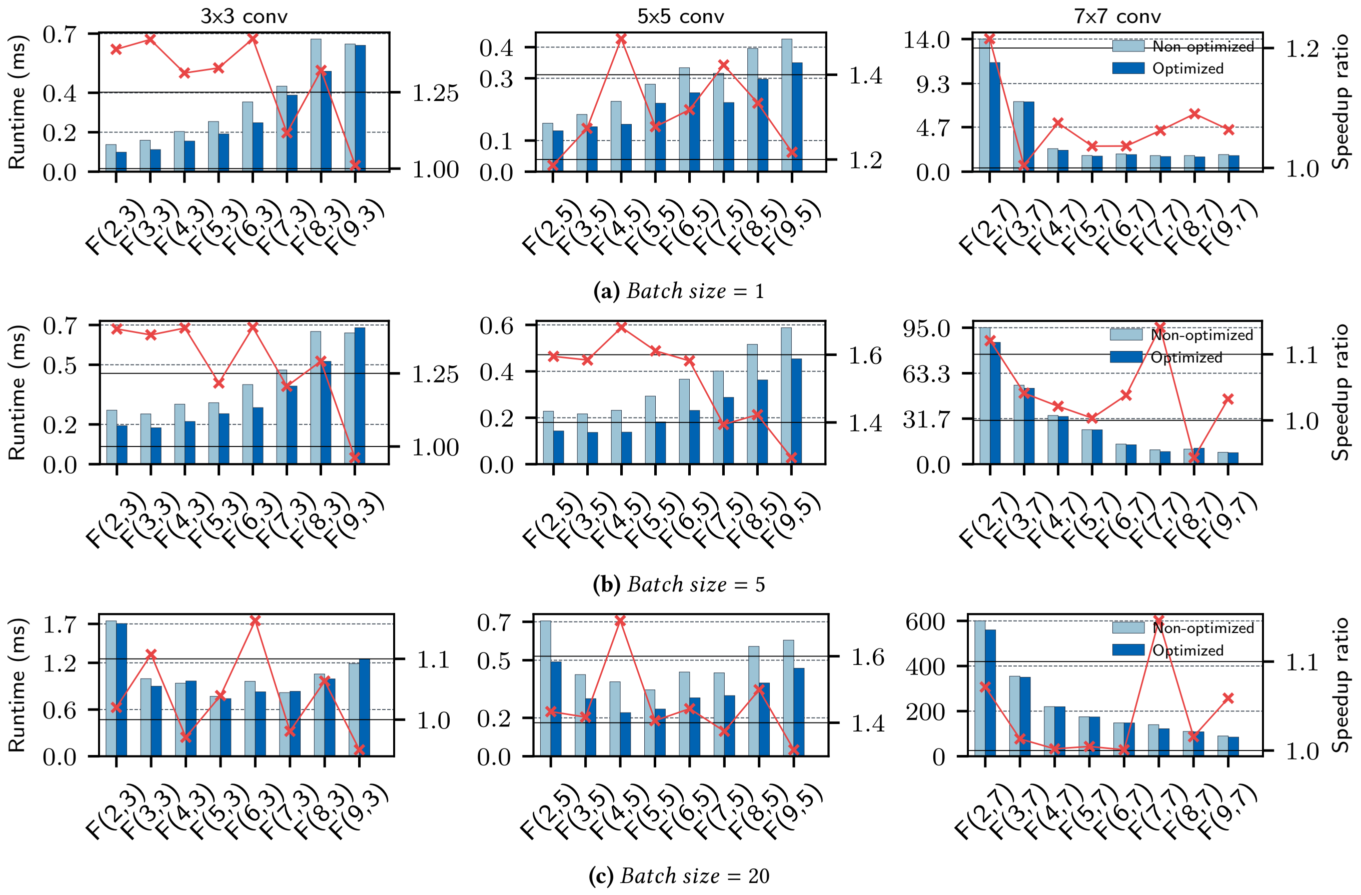


Figure 6. Comparing the runtimes of optimized and non-optimized Winograd convolutions $F(m^2, r^2)$, where $r \in \{3, 5, 7\}$, $m \in [2, 9]$, and the batch size $B \in \{1, 5, 20\}$.

We now present per-convolution-operation runtime results across three different hardware platforms to illustrate the efficiency and performance portability of our method. We sorted the operations by FLOP count, a reasonable proxy for their difficulty.

A runtime comparison of our method with cuDNN is given in Figure 7. It demonstrates the performance of our approach relative to the highly-tuned vendor library. We also included Boda's runtime in the absence of the Winograd convolution to display its impact on the performance of an inference engine. We observed that cuDNN's fused Winograd implementation only supports 3×3 convolutions. Our method, on the other hand, is more versatile and can generate efficient fused Winograd kernels for larger convolutions as well. The striped horizontal line in Figure 7 indicates the average speedup over cuDNN's Winograd convolutions. The results reveal that not only our method can often yield runtimes close to cuDNN's performance, but also can perform better than cuDNN, by up to 8.1× in some cases. However, cuDNN can achieve better runtimes for larger convolutions. We believe that this can be mainly attributed to more efficient matrix-multiplication routines.

Figure 8 compares the runtimes of our benchmark on the AMD GPU. We also included MIOpen runtimes as the baseline to show the performance of our method relative to the optimized AMD ConvNet library. The magenta striped line indicates the average speedup. Presumably benefiting from the highly-optimized MIOpenGEMM library, MIOpen performs better than our method for larger convolutions. However, in specific cases, we were able to outperform MIOpen Winograd implementation by up to a factor of 1.9. Together, Figures 7 and 8 illustrate that our method significantly improves the performance of the Boda inference framework. Additionally, we can achieve competitive performance compared with the vendor libraries on two different platforms. This observation confirms that our method is performance portable.

Furthermore, to validate the effect of auto-tuning on performance portability, we executed the code generated by our method with and without auto-tuning on our Mali G71 mobile GPU. This platform is entirely different from the previous two GPUs and usually requires an enormous amount of effort to achieve reasonable performance. Figure 9 illustrates the results of using the auto-tuner to select the right

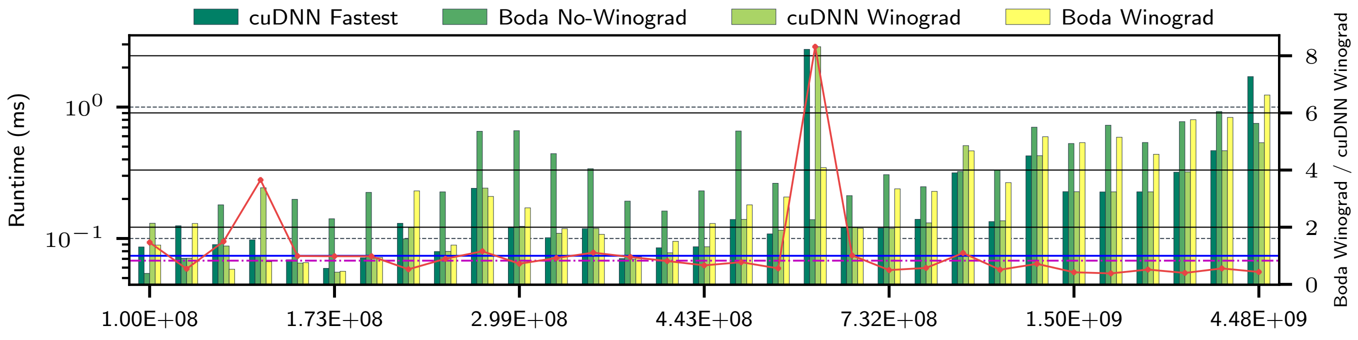


Figure 7. The runtime comparison of kernels generated using our method with cuDNN on Nvidia GTX 1080 Ti.

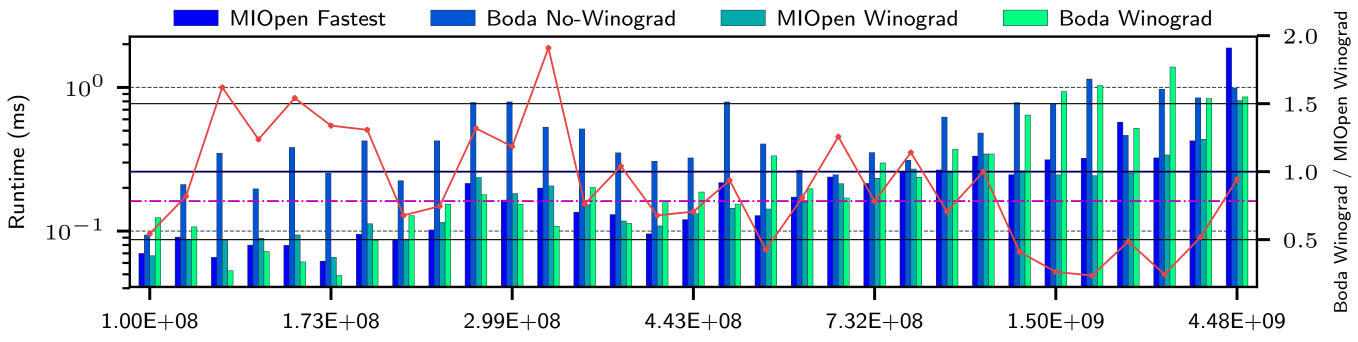


Figure 8. The runtime comparison of kernels generated using our method with MIOpen on AMD Radeon RX 580.

Winograd implementation and tuning parameters. We always used a non-fused implementation with $m = 2$, when auto-tuning is disabled. When auto-tuning is enabled, we can achieve a considerable speedup—on average, by a factor of 1.74. The red line shows the achieved speedup using auto-tuning for each convolution operation.

To compare the performance of our method with a well-known deep-learning library, we also added the Winograd convolution runtimes of the ARM compute library. The results in Figure 9 verify the importance of auto-tuning to achieve competitive results compared with other frameworks. Auto-tuning enabled us to find more efficient implementations and even surpass the performance of the ARM compute library for several convolution operations. We also noticed that the ARM compute library uses half-precision floating-point operations in matrix multiplications, which explains the reason for higher performance in other convolutions.

5 Related Work

Several studies have been conducted to reduce the arithmetic complexity of convolution operations [20]. Cong et al. [6] reduced the convolution runtime by up to 47% using the Strassen algorithm. Vasilache et al. [31] further reduced the computational complexity of convolutions using an FFT-based method. However, such a method is only practical in cases where the compute-to-memory ratio is high, and the cache size is limited. Thus, FFT convolutions are mostly used for convolutions with large image/filter sizes, and when the number of input/output channels is relatively small [35].

Soon after the seminal paper on Winograd convolution [16] appeared, the algorithm was integrated in popular deep-learning libraries such as Nvidia cuDNN, AMD MIOpen, and Intel MKL. Subsequently, several researchers attempted to make Winograd convolutions more accurate for larger kernel and input sizes [2, 32]. Our experiments show that the polynomial points selected by our method can produce slightly more accurate results. Further studies on the Winograd algorithm are mostly aimed at improving its performance on various hardware platforms, such as GPUs, CPUs, edge devices, and artificial-intelligence accelerators [3, 13, 34], reducing its computational complexity by leveraging sparse computations and parameter pruning [17, 18, 26].

To the best of our knowledge, existing methods for implementing efficient Winograd convolutions are geared toward a limited set of configurations (e.g., $F(2,3)$ and $F(4,3)$) and computing devices. Each study recommends a new set of optimizations, which are often bound to a specific hardware platform. For example, Xygkis et al. [34] attempted to

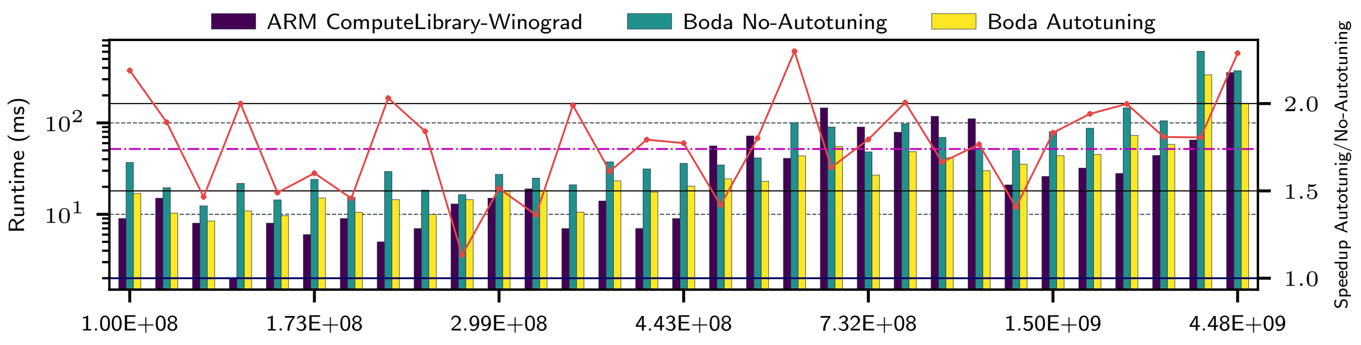


Figure 9. Winograd convolution performance with and without auto-tuning on Mali G71.

optimize the Winograd convolution on an Intel Movidius Myriad2 device. Such edge devices have limited power and memory capacity, and due to the high memory consumption of Winograd kernels, efficient memory management is essential. Therefore, the authors introduced optimization methods like data transfer management and data-representation optimization, which seems to be highly rewarding on Intel Myriad 2. However, they only evaluated their method for a single Winograd convolution.

In contrast to GPU devices, CPUs have access to a larger amount of memory. Such a feature enables inference frameworks to execute Winograd convolutions with higher dimensions and larger filters and output tile sizes. Three methods have been introduced to implement efficient Winograd kernels on CPUs [3, 13, 21]. Each of them suggests a different mixture of optimizations based on the target CPU. For instance, Jia et al. [13] demonstrated that their Winograd implementation can support n-dimensional convolutions. They employed various optimization techniques, including data layout optimization, an efficient SGEMM implementation, and transformation codelets for the efficient execution of Winograd operation on CPUs. Despite their successful attempt in accelerating Winograd, their method has been tested only on CPUs. Moreover, Jia et al. [13] claim that when the Winograd-internal tile size (i.e., α) is even, only input and filter transformations have a specific pattern that allows for further reduction in computational complexity. In contrast, we demonstrated that all three Winograd transformation matrices often contain a regular pattern, even when α is odd. Furthermore, none of the above-mentioned studies proposed a solution for making Winograd convolutions performance portable. Such a feature is crucial for inference frameworks, which aim to operate on various platforms.

To address performance portability, inference engines such as TVM [4], PlaidML [11], TensorFlow's XLA [1], and Glow [28] provide a platform to facilitate code generation and performance optimization. Among them, TVM framework [4] is the most comprehensive solution to run deep neural networks on a wide variety of hardware backends. TVM adopts the decoupled compute/schedule paradigm introduced in the Halide framework [27] and provides a domain-specific language for defining tensor operations and their optimization routines. Winograd convolution is also available in the TVM codebase. However, it is a non-fused implementation and uses predefined and hard-coded transformation matrices. We believe that integrating our symbolic analysis approach into the TVM's Winograd implementation can improve its versatility and runtime performance to a great extent.

6 Conclusion and Outlook

Winograd convolution is a promising method for reducing the computational complexity of convolution operations. However, if not appropriately implemented, the performance may be lower than expected. The overhead of the Winograd transformation steps can make it even inferior to the direct convolution. In this paper, we proposed a method based on symbolic computation to create minimal yet efficient recipes that replace the straightforward matrix multiplication method within Winograd transformations. Our empirical evaluation illuminated that choosing the right output tile size m, depending on the filter size, can significantly reduce the number of arithmetic operations while offering acceptable accuracy (e.g., $F(m = 6, r = 3)$, $F(m = 4, r = 5)$). To the best of our knowledge, this critical observation went unnoticed so far. Furthermore, we were able to generate performance-portable Winograd convolutions with the help of template meta-programming. Our runtime analysis shows that we can not only use the same Winograd meta-code to run on a multitude of GPU platforms, including a mobile GPU, but also compete with vendor ConvNet libraries, such as cuDNN, MIOpen, and the ARM compute library.

Finally, we believe that the proposed method can be used to target other platforms, such as CPUs, deep-learning accelerators, and dedicated inference engines (e.g., TVM [4]). To achieve higher speedups and better compatibility on new hardware, we plan to implement tunable BLAS routines tailored to Winograd multiplication steps.

Acknowledgment

We sincerely thank the EuroSys'20 reviewers for their insightful feedback and suggestions. We are particularly grateful to Wenjun Hu, who helped us significantly to improve the manuscript. This research has been supported by the Hessian LOEWE initiative within the Software-Factory 4.0 project and by the German Research Foundation (DFG) through the Program Performance Engineering for Scientific Software.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. In *Proc. of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Vol. 16. 265–283.

[2] Barbara Barabasz, Andrew Anderson, and David Gregg. 2018. Error analysis and improving the accuracy of Winograd convolution for deep neural networks. arXiv:1803.10986

[3] David Budden, Alexander Matveev, Shibani Santurkar, Shraman Ray Chaudhuri, and Nir Shavit. 2017. Deep tensor convolution on multicores. In *Proc. of the 34th International Conference on Machine Learning*. 615–624.

[4] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. TVM: An automated end-to-end optimizing compiler for deep learning. In *Proc. of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 578–594.

[5] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cuDNN: Efficient primitives for deep learning. arXiv:1410.0759

[6] Jason Cong and Bingjun Xiao. 2014. Minimizing computation in convolutional neural networks. In *Proc. of the International Conference on Artificial Neural Networks*. Springer, 281–290.

[7] Stephen A. Cook and Stål O. Aanderaa. 1969. On the minimum computation time of functions. *Trans. Amer. Math. Soc.* 142 (1969), 291–314.

[8] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. 2014. Training deep neural networks with low precision multiplications. arXiv:1412.7024

[9] Ross Girshick, Forrest Iandola, Trevor Darrell, and Jitendra Malik. 2015. Deformable part models are convolutional neural networks. In *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition*. 437–446.

[10] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. 2015. Deep learning with limited numerical precision. In *Proc. of the International Conference on Machine Learning*. 1737–1746.

[11] Intel. 2019. *PlaidML*. https://www.intel.ai/plaidml

[12] Sergey Ioffe and Christian Szegedy. 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. arXiv:1502.03167

[13] Zhen Jia, Aleksandar Zlateski, Fredo Durand, and Kai Li. 2018. Optimizing n-dimensional, Winograd-based convolution for manycore CPUs. In *Proc. of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 109–123.

[14] Andrej Karpathy, George Toderici, Sanketh Shetty, Thomas Leung, Rahul Sukthankar, and Li Fei-Fei. 2014. Large-scale video classification with convolutional neural networks. In *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition*. 1725–1732.

[15] Andrew Lavin. 2015. maxDNN: An efficient convolution kernel for deep learning with Maxwell GPUs. arXiv:1501.06633

[16] Andrew Lavin and Scott Gray. 2016. Fast algorithms for convolutional neural networks. In *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition*. 4013–4021.

[17] Sheng Li, Jongsoo Park, and Ping Tak Peter Tang. 2017. Enabling sparse Winograd convolution by native pruning. arXiv:1702.08597

[18] Xingyu Liu, Jeff Pool, Song Han, and William J Dally. 2018. Efficient sparse-Winograd convolutional neural networks. arXiv:1802.06367

[19] Jonathan Long, Evan Shelhamer, and Trevor Darrell. 2015. Fully convolutional networks for semantic segmentation. In *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition*. 3431–3440.

[20] Partha Maji and Robert Mullins. 2018. On the reduction of computational complexity of deep convolutional neural networks. *Entropy* 20, 4, 305.

[21] Partha Maji, Andrew Mundy, Ganesh Dasika, Jesse G. Beu, Matthew Mattina, and Robert Mullins. 2019. Efficient Winograd or Cook-Toom convolution kernel implementation on widely used mobile CPUs. arXiv:1903.01521

[22] Michael Mathieu, Mikael Henaff, and Yann LeCun. 2013. Fast training of convolutional networks through FFTs. arXiv:1312.5851

[23] Arya Mazaheri, Johannes Schulte, Matthew Moskewicz, Felix Wolf, and Ali Jannesari. 2019. Enhancing the programmability and performance portability of GPU tensor operations. In *Proc. of the 25th Euro-Par Conference, Göttingen, Germany (Lecture Notes in Computer Science)*, Vol. 11725. Springer, 213–226.

[24] Matthew W Moskewicz, Ali Jannesari, and Kurt Keutzer. 2016. A metaprogramming and autotuning framework for deploying deep learning applications. arXiv:1611.06945

[25] Cedric Nugteren. 2018. CLBlast: A tuned OpenCL BLAS library. In *In Proc. of the International Workshop on OpenCL (IWOCL)*. ACM, 5.

[26] Hyunsun Park, Dongyoung Kim, Junwhan Ahn, and Sungjoo Yoo. 2016. Zero and data reuse-aware fast convolution for deep neural networks on GPU. In *Proc. of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS)*. IEEE, 1–10.

[27] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *ACM Sigplan Notices* 48, 6 (2013), 519–530.

[28] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Garret Catron, Summer Deng, Roman Dzhabarov, Nick Gibson, James Hegeman, Meghan Lele, Roman Levenstein, et al. 2018. Glow: Graph lowering compiler techniques for neural networks. arXiv:1805.00907

[29] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. arXiv:1409.1556

[30] Andrei L Toom. 1963. The complexity of a scheme of functional elements realizing the multiplication of integers. In *Soviet Mathematics Doklady*, Vol. 3. 714–716.

[31] Nicolas Vasilache, Jeff Johnson, Michael Mathieu, Soumith Chintala, Serkan Piantino, and Yann LeCun. 2014. Fast convolutional nets with fbfft: A GPU performance evaluation. arXiv:1412.7580

[32] Kevin Vincent, Kevin Stephano, Michael Frumkin, Boris Ginsburg, and Julien Demouth. 2017. On improving the numerical stability of Winograd convolutions. In *Proc. of the International Conference on Learning Representations (ICLR)*.

[33] Shmuel Winograd. 1980. *Arithmetic complexity of computations*. Vol. 33. Siam.

[34] Athanasios Xygkis, Lazaros Papadopoulos, David Moloney, Dimitrios Soudris, and Sofiane Yous. 2018. Efficient Winograd-based convolution kernel implementation on edge devices. In *Proc. of the 55th Annual Design Automation Conference*. ACM, 136.

[35] Aleksandar Zlateski, Zhen Jia, Kai Li, and Fredo Durand. 2019. The Anatomy of Efficient FFT and Winograd Convolutions on Modern CPUs. In *Proc. of the ACM International Conference on Supercomputing (ICS)*. Association for Computing Machinery, New York, NY, USA, 414–424.