Optimizing N-Dimensional, Winograd-Based Convolution for Manycore CPUs

Zhen Jia*
Princeton University

Aleksandar Zlateski*
Massachusetts Institute of Technology

Fredo Durand
Massachusetts Institute of Technology

Kai Li
Princeton University

Abstract

Recent work on Winograd-based convolution allows for a great reduction of computational complexity, but existing implementations are limited to 2D data and a single kernel size of 3 by 3. They can achieve only slightly better, and often worse performance than better optimized, direct convolution implementations. We propose and implement an algorithm for N-dimensional Winograd–based convolution that allows arbitrary kernel sizes and is optimized for manycore CPUs. Our algorithm achieves high hardware utilization through a series of optimizations. Our experiments show that on modern ConvNets, our optimized implementation, is on average more than 3×, and sometimes 8× faster than other state-of-the-art CPU implementations on an Intel Xeon Phi manycore processors. Moreover, our implementation on the Xeon Phi achieves competitive performance for 2D ConvNets and superior performance for 3D ConvNets, compared with the best GPU implementations.

CCS Concepts • **Theory of computation → Massively parallel algorithms**; • **Computing methodologies → Neural networks**; • **Software and its engineering →** *Just-in-time compilers*;

Keywords convolution, winograd, vectorization, parallelization

ACM Reference Format:
Zhen Jia, Aleksandar Zlateski, Fredo Durand, and Kai Li. 2018. Optimizing N-Dimensional, Winograd-Based Convolution for Manycore CPUs. In *Proceedings of PPoPP '18: 23nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '18).* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3178487.3178496

*Both authors contributed equally to the paper

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
PPoPP '18, February 24–28, 2018, Vienna, Austria
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-4982-6/18/02…$15.00
https://doi.org/10.1145/3178487.3178496

1 Introduction

Convolutional neural networks (ConvNets) have emerged as a widespread machine learning method for many application domains, soon after the demonstration of its superior performance for classification and localization tasks in the ImageNet [20] competition in 2012 [33]. Convolutional layers are computationally intensive; they dominate the total execution time of modern, deep ConvNets [33, 40, 47, 49].

1.1 Related Work

Many efforts have been made to improve the performance of the convolutional primitives for CPUs [1, 15, 50, 56, 58] and GPUs [9, 16, 37, 51] or both [57]. An important class of improvements is to reduce the computations required for a convolution. Several efforts used FFT-based convolutions to reduce the required computations for GPUs [37, 51] and CPUs [56, 57].

Recently, Lavin et al. [34] proposed an algorithm based on the Winograd algorithm for minimal filtering, originally developed for fast computation of finite impulse response (FIR) filters [55]. The key idea of the Winograd–based convolution is similar to the one based on FFTs. The inputs and the kernels are first transformed. Then an element–wise multiplication, which is an equivalent problem to a matrix multiplication, is performed. An inverse transformation of the result yields the result of the convolution. Unlike the FFT–based convolution, where the element–wise multiplications are done in the complex domain, the Winograd–based convolution operates on real numbers, thus requiring fewer operations.

While the reduced number of operations should, in theory, speed up the computation, in practice, several challenges that make it hard to fully utilize the hardware resources on modern CPUs. **(1)** Winograd–based convolution might access the memory subsystem inefficiently, which may cost more than the time saved by performing fewer computations. **(2)** The Winograd-based convolution requires efficient matrix multiplications on tall and skinny matrices. Optimized libraries for matrix multiplication, such as Intel MKL and LIBXSMM [27, 28] do not achieve satisfactory performance

on such matrices [35, 54]. **(3)** The increasing of thread and data level parallelisms require new scheduling algorithms that minimize load balancing and synchronization overhead for manycore CPUs.

After Lavin et al. [34] demonstrated that Winograd based convolution can use fewer number of multiplications than FFT-based approach, especially for small 2D kernels (e.g. 3×3), Nervana [9] and Nvidia's cuDNN [16] had implemented Winograd based convolution for GPUs. CPU implementations were also provided by FALCON [1], LIBXSMM[10] and Intel MKL-DNN [2]. Budden et al. [15] further generalized the algorithm to arbitrary dimensions and kernel sizes, and proposed a CPU implementation. Unfortunately, all current Winograd–based implementations for CPUs perform well below what one would expect. As shown in this paper, in many cases, they under–performed more computationally expensive, but more optimized implementations including direct convolutions. Furthermore, currently available implementations only support 2D convolutions and single kernel size (3×3), which restricts the range of the application of Winograd–based convolution. 3D ConvNets are becoming more and more important, as they have been successfully applied to many fields [23, 30, 38, 39].

1.2 Contributions and Novelty

This paper has two main contributions. First, we present a new implementation for Winograd–based convolution for manycore CPUs. It is the first publicly available implementation that supports N–dimensional ConvNets with arbitrary kernel and tile sizes.

Second, to address the aforementioned challenges, we have proposed several optimizations for manycore CPUs, including:

- a custom storage data layout that allows efficient memory access by using only vector loads and stores, and a method to organize memory access patterns to minimize TLB misses,
- an utilization of streaming stores when the data will not be required in the near future, and amortization of the memory access overhead by interleaving computation with memory operations,
- a novel JIT code generator to generate optimal matrix multiplication routines for matrices of relevant sizes to maximize cache locality, and
- a static scheduling method for even parallel execution on the available cores to minimize synchronization overheads.

These optimizations have been proposed individually in the past; the novelty of our work is to combine them into a system of many parts, where the parts are designed to work well in conjunction with one another, and optimize the performance of the system as a whole.

We have tested our implementation with several representative ConvNets on an Intel manycore CPU (Xeon Phi 7210). Our results show that our implementation achieves on average 3x speedup, and in some cases more than 8x better performance than other existing Winograd–based implementations on the same hardware. Our custom JIT–compiled matrix multiplication achieves an average of 1.6x speedup and up to 2.5x for relevant matrices.

We have also compared the performance of our method with those optimized for modern GPUs. It achieved competitive performance for 2D ConvNets and much better performance for 3D ConvNets.

Our source code is publicly available at [3].

2 Background

2.1 Manycore CPUs

Manycore CPUs were evolved from multicore CPU designs [26] to provide applications with a high degree of parallel computing power using the shared–memory programming model. Earlier examples include Niagara processor [32] and Intel Larabee [46]. Recently, Intel introduced a Xeon Phi product line based on Many Integrated Core (MIC) architecture, including Knights Corner [17] and Knights Landing (KNL) [48].

The rationale for manycore CPUs is to allow certain applications that run well on multicore CPUs to scale up performance with little or no modification. However, to fully take advantage of the power of a manycore CPU, one must be aware of its key architectural features compared to general-purpose multicore CPUs:

- a large number of simpler CPU cores running on lower frequencies,
- 2 levels of cache instead of a sophisticated 3 or 4 levels,
- wide vector units for each core to deliver high floating point performance, and
- high throughput, multi–channel memory.

The optimization methods presented in this paper have considered these features. Our implementation was configured for the latest Intel Xeon Phi (Knights Landing) processor, which has the following specific features:

- It has 64 to 72 x86 CPU cores depending on different models. Each core supports 4 hardware hyper–threads.
- Each core contains a two–wide, out–of–order pipeline, and two vector processing units (VPUs) that support 512–bit Advanced Vector Extension instruction set (AVX–512). It supports a scalar–vector Fused–Multiply–Add (FMA) instruction, where each value in one of the 32 vector registers is multiplied by a scalar from memory and the result is accumulated to a value in another register. Each core is thus capable of 64 single precision FLOPs per cycle – each of the two VPUs perform an FMA on 16 floating points.
- Each core can issue up to two memory operations per cycle, either load or store; the operations can be executed and retired out–of–order.

- Each core has a 32KB write–back L1 data cache and supports two simultaneous 512–bit reads and one write. A pair of two cores share a 1MB L2 cache.
- It supports an explicit prefetch instruction, to either L1 or L2. For non–temporal memory accesses, streaming store instructions can be used to write data directly into main memory to eliminate cache pollution and consume less bandwidth [29].
- It is equipped with High–Bandwidth Memory (HBM) based on the multi–channel dynamic random access memory (MCDRAM), capable of delivering up to 5× performance compared to DDR4 memory on the same platform [11].

The challenge is to fully utilize the hardware's peak performance.

2.2 Winograd Algorithm

Winograd proposed an algorithm for efficient computing of finite impulse response (FIR) filters in the signal processing field [55].

Computing m outputs with an r–tap FIR filter, which we denote as $F(m, r)$, requires $m \times r$ multiplications through the direct method. The Winograd algorithm reduces the number of multiplications to $m + r - 1$.

Using $F(2, 3)$ as an example, we explain how such reduction is achieved. For an input vector $d = (d_0, d_1, d_2, d_3)$ and filter $g = (g_0, g_1, g_2)$, the Winograd algorithm transforms the input data and the filter to $j = (j_0, j_1, j_2, j_3)$ and $h = (h_0, h_1, h_2, h_3)$ respectively via

$$\begin{array}{l} j_0 = d_0 - d_2,\ h_0 = g_0 \\ j_1 = d_1 + d_2,\ h_1 = \frac{g_0+g_1+g_2}{2} \\ j_2 = d_2 - d_1,\ h_2 = \frac{g_0-g_1+g2}{2} \\ j_3 = d_1 - d_3,\ h_3 = g_2 \end{array} \tag{1}$$

Then, it performs element wise multiplications between j and h as:

$$\begin{array}{l} c_0 = j_0 \times h_0,\ c_1 = j_1 \times h_1 \\ c_2 = j_2 \times h_2,\ c_3 = j_3 \times h_3 \end{array} \tag{2}$$

That is, the final result $y = (y_0, y_1)$ is then computed via:

$$y_0 = c_0 + c_1 + c_2,\ y_1 = c_1 - c_2 - c_3 \tag{3}$$

Winograd [55] showed that

$$F(2,3) = \begin{bmatrix} d_0 & d_1 & d_2 \\ d_1 & d_2 & d_3 \end{bmatrix} \begin{bmatrix} g_0 \\ g_1 \\ g_2 \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \end{bmatrix} \tag{4}$$

The response y is computed as $y = \mathcal{A}\left[(\mathcal{G}g) \odot (\mathcal{B}d)\right]$, where $\odot$ represents element–wise multiplication and

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

$$\mathcal{G} = \begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 1 \end{bmatrix}, \mathcal{B} = \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix} \tag{5}$$

The element–wise product in Eqn. 2 requires $m + r - 1 = 2 + 3 - 1 = 4$ multiplications, whereas the direct method needs $m \times r = 2 \times 3 = 6$ multiplications.

The Winograd method can yield a dramatic reduction in the number of required operations when a signal is convolved with multiple kernels and when multiple signals are convolved with a single kernel; both of which are properties of the convolutional layers of ConvNets.

3 N–Dimensional Convolutional Layers

This paper focuses on convolutional layers which dominate the ConvNet computation time. A convolutional layer transforms an input tuple of C images into an output tuple of C' images. A batch of B inputs yielding a batch of B outputs is processed at the time via

$$I'_{b,c'} = \sum_{c=1}^{C} I_{b,c} * W_{c'c} \tag{6}$$

where $*$ represents convolution, C and C' denote the number of input/output images (also called channels or feature–maps). $I_{b,c}$ and $I'_{b,c'}$ are the (arbitrary dimensional) input and output images of the b–th batch.

3.1 1D Winograd Convolution

In the 1D case, the $F(m, r)$ FIR filter operation can be used to compute 1D convolution of a 1D image (vector) of size $T = m+r-1$ with a kernel size r, producing a result vector of size m. Due to Winograd's numerical instability [15, 34, 53], T can only take very small values. For larger vectors the overlap–add method (OLA) [43] is used. A vector of size l, which is greater than T, is divided into $\lceil (l - r + 1)/m \rceil$ sub–vectors (tiles) of size $T = m + r - 1$ with $r - 1$ elements of overlap. If necessary, the vector is zero padded.

The tile $I^i_{c'}$ (i^{th} tile of the c'^{th} output channel) is computed using the Winograd algorithm via

$$\begin{aligned} I^i_{b,c'} &= \sum_{c=1}^{C} \mathcal{A}\left[(\mathcal{G}W_{c,c'}) \odot (\mathcal{B}I^i_{b,c})\right] \\ &= \mathcal{A}[\sum_{c=1}^{C} (\mathcal{G}W_{c,c'}) \odot (\mathcal{B}I^i_{b,c})] \end{aligned} \tag{7}$$

Note that, in Eqn. 7, the inverse transform $\mathcal{A}$ is applied to the sum, greatly reducing the number of required operations.

3.2 N–Dimensional Winograd Convolution

The basic Winograd algorithm has been extended to N–D convolutions, where the input/output images (I_c and $I_{c'}$), as well as the kernels ($W_{c,c'}$) are N–dimensional [15]. We adopt the notation from Budden et al. [15], where $F(m_1 \times m_2 \cdots \times m_n, r_1 \times r_2 \times \cdots \times r_n)$ represents Winograd FIR filter operation on an N–D signal with an N–D filter.

As in the 1D case, the convolution of large images is performed using the overlap–add method: the input N–D images are divided into N–D sub–images (tiles), of size $(m_1 + r_1 -$

$1) \times (m_2 + r_2 - 1) \times \cdots \times (m_n + r_n - 1)$, with $r_d - 1$ overlap along each dimension d. The result of the output tile $I'^{(\vec{i})}_{c'}$ of size $m_1 \times m_2 \times \cdots \times m_n$ is computed via

$$I'^{(\vec{i})}_{b,c'} = \left[\sum_{c=1}^{C} (W_{c,c'} \times_n^N \mathcal{G}) \odot (I^{(\vec{i})}_{b,c} \times_n^N \mathcal{B})\right] \times_n^N \mathcal{A}$$
$$= \left[\sum_{c=1}^{C} \mathcal{W}_{c,c'} \odot I^{(\vec{i})}_{b,c}\right] \times_n^N \mathcal{A} = \mathcal{I}'^{(\vec{i})}_{b,c'} \times_n^N \mathcal{A} \quad (8)$$

Here, $\vec{i}$ is the tile coordinate $\langle i_1, i_2, \ldots, i_n \rangle$. The operation $X \times_n^N Y$ is short for $X \times_1 \times_2 \cdots \times_n Y$, where $\times_n$ represents tensor–matrix mode–n multiplication as defined in [15, 31].

3.3 Reduction to Matrix Multiplications

In Eqn. 8, all of $\mathcal{W}_{c,c'}$, $\mathcal{I}^{(\vec{i})}_{b,c}$ and $\mathcal{I}'^{(\vec{i})}_{b,c'}$ have the same size of $(m_1 + r_1 - 1) \times (m_2 + r_2 - 1) \times \cdots \times (m_n + r_n - 1)$. Each image element $\vec{j}$ of $\mathcal{I}'^{(\vec{i})}_{b,c'}$ is computed as

$$\mathcal{I}'^{(\vec{i})}_{b,c'}\left[\vec{j}\right] = \sum_{c=1}^{C} \mathcal{I}^{(\vec{i})}_{b,c}\left[\vec{j}\right] \cdot \mathcal{W}_{c,c'}\left[\vec{j}\right] \quad (9)$$

For a fixed value of $\vec{j}$ the equation above resembles a matrix–matrix multiplication. The left hand side can be represented as a matrix with c' columns, and the number of rows equal to the product of the batch size B and the number of N–D tiles. The matrix representing $\mathcal{I}^{(\vec{i})}_{b,c}\left[\vec{j}\right]$ has c columns, and the same number of rows. Finally, the matrix representing $\mathcal{W}_{c,c'}\left[\vec{j}\right]$ has c' rows and c columns.

When training modern ConvNets, a relatively large batch size B is used (32 or 64), and also the images typically consist of a very large number of tiles (as the values of m_i are relatively small compared with the N–D image dimensions). In such case, matrix multiplications are performed on very tall and skinny matrices.

4 Algorithm and Optimizations

This section gives an overview of our algorithm and presents several optimization methods. Fig. 1 shows the overview of our algorithm which consists of the three stages of the Winograd–based convolution: input image and kernel transformations, matrix multiplications, and the inverse transformation of the result.

For clarity, we use 3D images and kernels to describe our algorithm and several optimization methods for many-core CPUs. The generalization to N–dimensional images is straightforward. The algorithm presented below is for general $F(m_D \times m_H \times m_W, r_D \times r_H \times r_W)$.

4.1 Data Layout

Before giving the algorithm details, we describe how the data are stored in memory. The design of our data layout is guided by the following two principles:

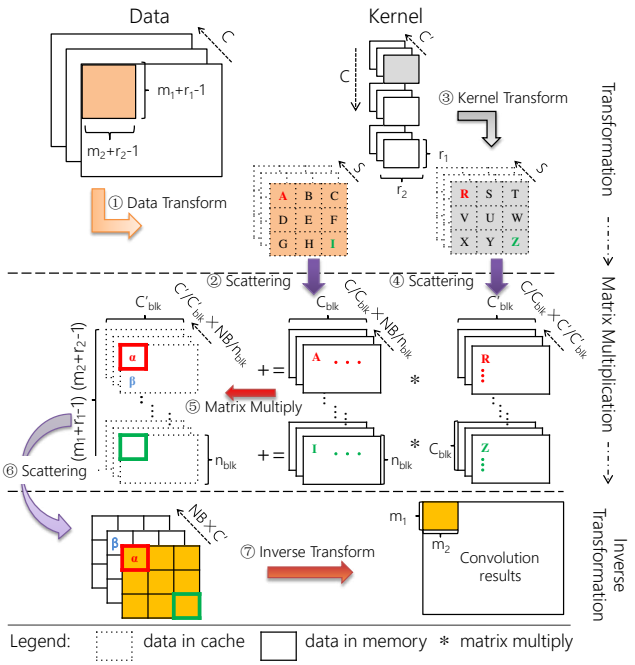

Figure 1. The overview of our algorithm using 2D images as an example.

1. To allow for using only aligned vector loads/stores, which enables easy vectorization
2. The computations should access a relatively small range of memory – to minimize cache and TLB misses.

For images and kernels, we generalize the data layout proposed in [29, 58] to N–dimensional images. Each image is stored in a multi–dimensional row–major array. The data is 64-byte aligned so as to facilitate the consecutive and aligned memory operations.

A batch of B tuples of C images, each of size $D \times H \times W$ is stored in memory as an array of size $B \times \lceil C/S \rceil \times D \times H \times W \times S$, where S represents the width of the vector register, which is the number of single–precision floating point numbers that fit in a vector register (16 for AVX-512). In this paper, we assume that the number of input and output channels is divisible by $S = 16$, which is true for all modern ConvNets.

Kernels are stored as an array of size $C \times \lceil C'/S \rceil \times r_D \times r_H \times r_W \times S$, where C/C' represents the number of input/output channels, and r_D, r_H, r_W are the kernel dimensions.

Tbl. 1 shows the memory locations (in their corresponding arrays) where each image and kernel elements as well as their transformed data structures are stored. The details are discussed below.

It is important to note that in ConvNets, the output of one layer is the input to the next layer thus no data reshuffling between layers is necessary.

Table 1. A translation table of our custom data layout, depicting where each element of images/kernels and their transforms are stored in memory. $I_{b,c}^{(d,h,w)}$ represents a pixel value at location $\langle d, h, w\rangle$ of the image $I_{b,c}$, and $\mathcal{I}_{b,c}^{n,(t_d,t_h,t_w)}$ represents the value at location $\langle t_d, t_h, t_w\rangle$ of the transformed n–th tile of the image $I_{b,c}$. Here, $n' = bN + n$, and $t = t_d T_h T_w + t_h T_w + t_w$

Variable	Element	Location in the multi–dimensional array (stored in memory)	Comment
Input images	$I_{b,c}^{(d,h,w)}$	$I[b][c/S][d][h][w][c \mod S]$	–
Transformed inputs	$\mathcal{I}_{b,c}^{n,(t_d,t_h,t_w)}$	$\mathcal{I}[n'/n_{blk}][c/C_{blk}][t][n' \mod n_{blk}][c \mod C_{blk}]$	T matrices of size $NB \times C$
Kernels	$W_{c,c'}^{(d,h,w)}$	$W[c][c'/S][d][h][w][c' \mod S]$	–
Transformed kernels	$\mathcal{W}_{c,c'}^{(t_d,t_h,t_w)}$	$\mathcal{W}[c/C_{blk}][c'/C'_{blk}][t][c \mod C_{blk}][c' \mod C'_{blk}]$	T matrices of size $C \times C'$
Transformed outputs	$\mathcal{I}'^{n,(t_d,t_h,t_w)}_{b,c'}$	$\mathcal{I}'_{tmp}[n'/n_{blk}][c'/C'_{blk}][t][n' \mod n_{blk}][c' \mod C'_{blk}]$	T matrices of size $NB \times C'$
		$\mathcal{I}'[b][c'/S][n][t_d][t_h][t_w][c' \mod S]$	–
Output images	$I'^{(d,h,w)}_{b,c'}$	$I'[b][c/S][d][h][w][c' \mod S]$	–

4.2 Stage 1 – Input and Kernel Transforms

In the first stage, the transforms $\mathcal{I}$ and $\mathcal{W}$ are computed from images I and kernels W. For the input images, this is done by first subdividing each 3D image into overlapping 3D **tiles** of size $T_D \times T_H \times T_W = (m_D + r_D - 1) \times (m_H + r_H - 1) \times (m_W + r_W - 1)$. The tiles have overlap of $(r_D - 1)$, $(r_H - 1)$ and $(r_W - 1)$ along the three dimensions. There are a total of N tiles per image, $N = N_D \times N_H \times N_W = \lceil (D - r_D + 1)/m_D \rceil \lceil (H - r_H + 1)/m_H \rceil \lceil (W - r_W + 1)/m_W \rceil$. Each tile has $T = T_D T_H T_W$ elements.

We perform tensor–matrix mode–n multiplication of each tile with $\mathcal{B}$ to obtain the transform (of same size) and store data into $\mathcal{I}$ (operations ① and ② in Fig. 1. Optionally, the kernel transforms, $\mathcal{W}$ are also computed (operation ③). The $C \times C'$ kernels of size $r_D \times r_H \times r_W$ are similarly transformed and stored into $\mathcal{W}$ (operation ④). Each kernel is transformed by tensor–matrix mode–n multiplication with $\mathcal{G}$. The kernel transforms have the same size as the transforms of the image tiles.

The main unit of computation is a transformation of S tiles of size $T_D \times T_H \times T_W$ for images and $r_D \times r_H \times r_W$ for kernels, at the time using generated **vectorized** codelets. These are tiles from S adjacent channels of the same batch located at the same offset in the input images. The data layout of our input images and kernels allows for using only aligned vector loads from memory and vector instructions to easily operate on S tiles at the time.

4.2.1 Transformation codelets

In order to efficiently support arbitrary $F(m, r)$, we created a codelet generator that creates an efficient tensor–matrix mode–n multiplication code of an arbitrary N–D image and the matrices $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{G}$. The generated code operates elements from $S = 16$ images at a time, multiplying each of them with one of the three transformation matrices. This is achieved by using the vector registers and Intel intrinsics [7].

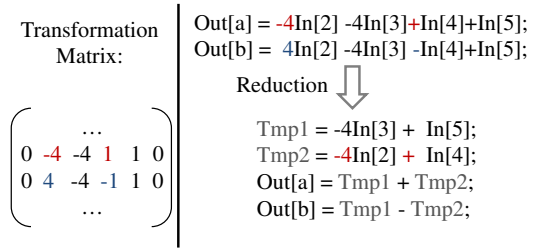

Figure 2. Reducing computation in transformations. Computing the values of the two output variables is reduced from 6 to 4 FMA instructions, and the latency is reduced from 18 to 12 cycles (assuming 6 cycles latency of FMAs on KNL).

We used Wincnn [13] to generate the transformation matrices from which a templated C++ code is created. The template parameters are m, r, and the strides of the input and output images (mapping images to memory locations). As we assume that m and r and image/kernel sizes are known during compile time, the templated c++ codelets allow for zero–overhead computation.

Matrices $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{G}$ are sparse; the percentage of non–zero elements decreases as m and r increase [15]. Additionally, when $m + r - 1$ is even, some of the rows in $\mathcal{B}$ and $\mathcal{G}$ have a pattern that allows for further reduction in computational complexity and compute latency. An example is shown in Fig. 2. Our codelets are designed to produce code with the minimal number of operations.

Additionally, our codelets store the results using non–temporal, streaming vector stores, as the transformation results will not be used in the near feature (until the next stage). This saves memory bandwidth, avoids cache pollution, and is crucial for optimal performances.

Image and kernel transforms data layout The transformed images and kernels are stored in memory in arrays of size $(NB/n_{blk}) \times (C/C_{blk}) \times T \times n_{blk} \times C_{blk}$ and $(C/C_{blk}) \times (C'/C'_{blk}) \times T \times C_{blk} \times C'_{blk}$ respectively (Tbl. 1).

Here, $6 \leqslant n_{blk} \leqslant 30$ and $32 \leqslant C_{blk}, C'_{blk} \leqslant 512$, with additional constraints of both C_{blk} and C'_{blk} being divisible by 16, and $C_{blk} \times C'_{blk} \leqslant 128^2$. The values of n_{blk}, C_{blk} and C'_{blk} are chosen for optimal matrix multiplication performance of the second stage, and will be discussed in more detail in the following paragraphs.

With this layout, the codelets access a relatively small range of memory when storing the results of the transformation. For the images, they access the range of size $T \times n_{blk} \times C_{blk}$, which is the scattering range of ②, and for the kernels the range of size $T \times C_{blk} \times C'_{blk}$ (scattering range of ④); thus allowing for minimal TLB misses.

Inference only For the inference only computation (applying pre–trained ConvNet to new data), we can omit the kernel transformations, by memorizing the transformed data, as the values of $W_{c,c'}$ do not change. As our experiments demonstrate, in certain cases, this can save a significant amount of time.

4.3 Stage 2 – Matrix Multiply

At the second stage (operation ⑤), the transforms of the output images I' are computed using matrix multiplications. Each of T matrices of size $NB \times C$ obtained by transforming the inputs are multiplied by the corresponding matrix (of transformed kernels) of size $C \times C'$ obtaining T matrices of size $NB \times C'$. This can be accomplished by T matrix multiplications $X = U * V$.

Our method to implement matrix multiplication routines is to employ the matrix blocking strategy to reduce cache misses [22, 26]. We subdivide U into sub–matrices of size $n_{blk} \times C_{blk}$, the matrix V into ones of size $C_{blk} \times C'_{blk}$, and the matrix X into $n_{blk} \times C'_{blk}$, as shown in Fig. 3. Each sub–matrix $X_{i,j}$ is computed via

$$X_{i,j} = \sum_{k=1}^{C/C_{blk}} U_{i,k} * V_{k,j} \tag{10}$$

The computation is performed in an order that maximizes data reuse in cache: for each sub–matrix $V_{k,j}$, we loop over $1 \leqslant i \leqslant NB/n_{blk}$ and perform $X_{i,j} = \beta X_{i,j} + U_{i,k} * V_{k,j}$. Here β is 0 during the first loop over i, and 1 otherwise. As U and X are tall and skinny [1, 54], we perform many multiplications with the same $V_{k,j}$, which stays in L2 cache after the first use.

The novel aspect of our approach is to perform in–cache computation of the sub–matrices of U and V using our custom JIT–compiled matrix multiplication primitive. It is optimized for batched multiplications (with the same small $\hat{V}$). It efficiently performs matrix–matrix multiplications of matrices of size $n_{blk} \times C_{blk}$ and $C_{blk} \times C'_{blk}$ ($X_{i,j} = \beta X_{i,j} + U_{i,k} * V_{k,j}$), while also pre–fetching $U_{i+1,j}$ and $X_{i+1,j}$ to L2 cache.

4.3.1 JIT–compiled batch matrix multiplication

Using JIT–compiled primitives allows for zero run–time overhead. As the matrix sizes are known during instantiation (compilation) time, we can optimally unroll loops, and pre–compute all memory access offsets. Producing assembly code allows for a fine grain control of the CPU, as well as the size of the generated binary code. Our custom primitive computes $\hat{X} = \beta\hat{X} + \hat{U} * \hat{V}$, where $\hat{X}$, $\hat{U}$ and $\hat{V}$ are sub–matrices of X, U and V with the assumption that all of $\hat{U}$, $\hat{V}$ and $\hat{X}$ are in L2 cache most of the time. For each size of $n_{blk}, C_{blk}, C'_{blk}$ and β an assembly implementation is generated on demand, which is then compiled to a shared library, and loaded into the shared memory for use. Only multiples of $S = 16$ for C_{blk} and C'_{blk} are supported.

We have implemented another level of "blocking" for registers. Sub–matrices of $\hat{X}$ of size $n_{blk} \times S$ are computed at the time, while keeping the intermediate results in the vector registers. Our algorithm requires 2 auxiliary registers, thus limiting n_{blk} to 30 (there are 32 available AVX–512 vector registers).

For ease of explanation, let $C_{blk} = R \times S$ and $C'_{blk} = Q \times S$. When computing the sub–matrix of $\hat{X}$ containing all n_{blk} rows, and S columns $[qS, qS + S)$, we first load the current values of $\hat{X}$ into n_{blk} vector registers, each holding values of one row and S columns of $\hat{X}$ sub–matrix. When β is 0, the registers are zero initialized.

The operation $\hat{X} = \hat{X} + \hat{U} * \hat{V}$ is computed by first looping over i (the S columns) and then over j (the n_{blk} rows) of $\hat{U}$ and performing a scalar–vector FMA of $\hat{U}_{j,i}$ with the i–th row of $\hat{V}$ and accumulating the result into the i–th row of $\hat{X}$ that is held in one of the vector registers. During each loop over i one additional vector load to register is performed ($(i+1)$–th row of $\hat{V}$), to allow for in–register operations in the next iteration of i. Also, up to 4 pre–fetches (from $L2$) to $L1$ are issued, fetching elements of $\hat{U}$ and $\hat{V}$ that will soon be used. The load and the pre–fetch instructions are interleaved with the FMAs. Both loops over i and j are fully unrolled into assembly instructions. Operations of an unrolled loop over j for $i = 3$ is shown on Fig. 4.

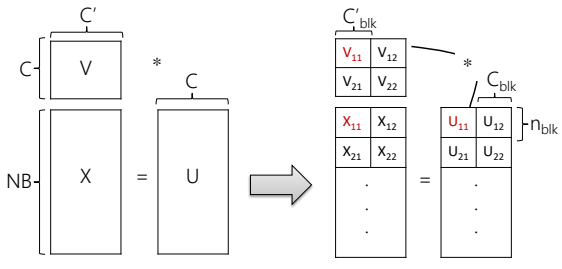

Figure 3. Decomposing matrix multiplication of large matrices to blocked, batched multiplication of smaller ones.

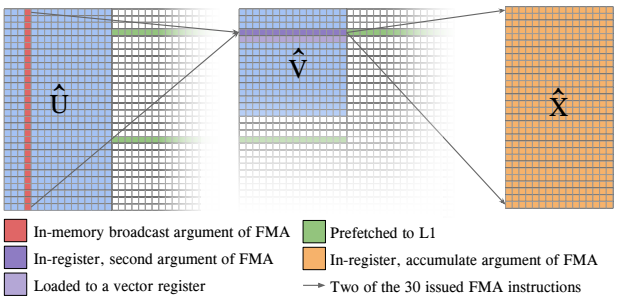

Figure 4. FMA and prefetch operations of our JIT-compiled matrix multiplication primitive in 1 of 16 unrolled loops.

The resulting rows are then stored back to $\hat{X}$. When storing each of the n_{blk} rows, we pre–fetch the data from the same locations in next two matrices to be multiplied by $\hat{V}$ to L2 cache. The computation then consists of $\hat{X}$ sub–matrix multiplications, where, each sub–matrix of $\hat{U}$ of size $n_{blk} \times S$, $S = 16$, is multiplied with a sub–matrix of $\hat{V}$ of size $S \times S$ and accumulated to $\hat{X}$. Additionally, to reduce the instruction sizes, we use X86 SIB (scale index base).

Scattering matrix multiply results After all the result of $\hat{U} * \hat{V}$ had been accumulated to $\hat{X}$, it is scattered back to appropriate locations in $\mathcal{I}'$ for the next stage's use (operation ⑥ in Fig. 1). This allows the next stage to access consecutive memory, avoiding expensive gathering operations, with the expense of some memory overhead (we use $\mathcal{I}'_{tmp}$ for intermediate results in our cache blocked loop and $\mathcal{I}'$ for final matrix–multiply results).

The scattering is performed inside our JIT primitives using non–temporal streaming stores, in order to mitigate cache misses and avoid cache evictions. As the primitives are pretty compute intensive, possible TLB miss overhead due to the large memory access stride is amortized out. Scattering data inside our JIT primitives increased the overall speed by more than 20%.

4.3.2 Choosing optimal blocking sizes

For optimal performances of our convolutional primitives, the blocking parameters n_{blk}, C_{blk} and C'_{blk} are empirically determined during instantiation time.

While $V_{k,j}$ is kept in L2 cache, each of the U and X sub–matrices have to be fetched from main memory, and the results stored back. Each operation $\hat{X} = \beta\hat{X} + \hat{U} * \hat{V}$ thus requires $\beta n_{bkl} C'_{blk} + n_{Blk} C_{blk}$ loads, and $n_{bkl} C'_{blk}$ stores.

The compute–to–memory ratio is then:

$$\frac{2n_{blk}C_{blk}C'_{blk}}{n_{blk}((\beta+1)C'_{blk} + C_{blk})} = \frac{2C_{blk}C'_{blk}}{(\beta+1)C'_{blk} + C_{blk}} \tag{11}$$

Ideally, we would like this ratio to be as high as possible (higher than the machine capabilities). When $C_{blk} = 128$ and $C'_{blk} = 128$, and $\beta = 1$, this ratio is 85.33, which is much higher than the compute–to–memory ratio of the Xeon Phi processor of 45 (approximately 4.5 TFLOPS computing of floating point, and approximately 400 GBytes/s = 100 GFloats/s). In this case, $\hat{V}$ requires 64 KBytes of L2 cache, allowing for additional 448 or 192 KBytes for two copies of $\hat{U}$ and $\hat{X}$ when 1 or 2 threads per core are used respectively.

When $C_{blk} = C'_{blk} = 64$, the ratio is 42.67, far below the capabilities of the processor – implying that the performance will be limited by memory.

Our method requires C and C' to be divisible by C_{blk} and C'_{blk}, but allows for an arbitrary n_{blk}. When NB is not divisible by n_{blk}, we pad the last sub-matrix of U. The choice of n_{blk} determines the memory required for $\hat{U}$ and $\hat{X}$. Lower values of n_{blk} will leave more memory space for $\hat{V}$ – thus higher C_{blk} and C'_{blk}. Being so delicately inter–dependent, we take the strategy of FFTW [21] and determine the values of n_{blk}, C_{blk} and C'_{blk} as well as how many threads to use per core empirically for each particular layer shape.

Determining optimal values of the parameters takes a relatively small amount of time and allows for saving the optimal parameters in a wisdom file. However, we limit the search to $6 \leqslant n_{blk} \leqslant 30$ – as less than 6 wouldn't allow for full utilization of the vector units due to the FMA latencies, and 30 is the maximal amount of registers available. We limit the sizes of C_{blk} and C'_{blk} to be greater than 64 when possible – for a good compute–to–memory ratio; but smaller than 512, with $C_{blk} \times C'_{blk} \leqslant 128^2$ due to L2 memory limitations.

4.4 Stage 3 – Inverse Transformation

In the final stage, the output images I' are computed from $\mathcal{I}'$. This is accomplished using inverse transformation codelets that are generated in the same manner as the image and kernel transformation ones. Each codelet transforms S tiles at the time and stores the result to appropriate locations in I'. A total of BNC'/S transformations are performed.

The previous stage has ensured that each transformed output occupies a contiguous chunk of memory, which ensures as fast memory access and as fewer TLB misses as possible.

Memory overhead Our algorithm requires auxiliary memory for temporary values (image/kernel transforms and matrix multiplication results). A memory buffer capable of fitting all of $\mathcal{I}$, $\mathcal{W}$, $\mathcal{I}'_{tmp}$ and $\mathcal{I}'$ is required.

While the size of the auxiliary buffer can be a couple of times larger than the memory required for storing the computed images, the same memory buffer can be reused for the computation of each layer. As modern ConvNets contain a large number of layers, the size of the auxiliary buffer represents a small fraction of the total memory required for a ConvNet.

4.5 Parallelization Through Static Scheduling

Our approach parallelizes each of the three stages of our algorithm using static scheduling.

The main idea is to pre–assign work for each of the available cores/threads. To achieve optimal performances, each core is assigned the same amount of computation with the same memory access patterns. The work is then executed using a single fork–join. Ideally, all the threads would start and finish the work at the same time, thus not having any core idling at any point in time.

To achieve this goal, we have generalized the static scheduling approach proposed in [58]. Our method is to represent the total work as a D–dimensional grid of equal tasks operating on different data. For instance, in the first stage of our algorithm, we need to transform a batch of B tuples of C images, each containing $N_D \times N_H \times N_W$ tiles. The computation performed on each tile is the same. We can represent this as a grid of size $B \times C \times N_D \times N_H \times N_W$ of *tile transformation* tasks. Here B is the most significant dimension, and N_w the least significant. The grids are created in a way that benefits more from executing tasks along less significant dimensions on the same core. In this case, transforming two adjacent tiles along $_W$ benefits from cache reuse, as the two tiles have overlapping elements.

Our scheduling algorithm uses the following recursive way of dividing an arbitrary D–dimensional grid of tasks $(P_1 \times P_2 \times \cdots \times P_D)$ among a set of K threads (one or more threads per core). In the base case, when $|K| = 1$, it schedules all tasks to that particular thread. Otherwise, it finds the most significant dimension d, such that the largest common divisor $x_d = GCD(P_d, |K|)$ is greater than one. The algorithm slices the grid along d into x_d equal sub–grids, and divides the set of threads K into x_d sub–sets with $|K|/x_d$ threads. It then solves each of the x_d sub–problems recursively. In the case when no GCD is greater than one, it divides the grid along the dimension d with the largest P_d (largest dimension size) as equally as possible. This assigns slightly more work to some threads than to others.

At the first stage, the grid size is $B \times (C/S) \times N_D \times N_H \times N_W$, as our transformation tasks (codelets) operate on S tiles at the time. Since the number of threads, as well as the B (batch size) and C (number of input channels) are typically powers of two, this scheduling approach will nearly always evenly divide the work by just slicing along more significant dimensions.

At the second stage, T matrix multiplications are performed. The size of the resulting matrices is $NB \times C'$, where each resulting sub–matrix of size $n_{blk} \times C'_{blk}$ can be computed separately as described above. Here, the grid of sub–tasks has size $T \times (C'/C'_{blk}) \times (NB/n_{blk})$. (NB/n_{blk}) is set as the least significant dimension, as each of the (NB/n_{blk}) sub–matrices are multiplied with the same matrix (which is then kept in cache) when all multiplications are performed by a single thread.

Finally, the third stage performs $B \times N \times C'$ inverse transformations (B tuples of C' images each having N tiles to be transformed). Here, there's no memory overlap, so each transformation operates on a disjoint sub–set of memory. In this case, the algorithm simply forms 1–dimensional grid of tasks of size BNC'/S – each task is computed by our codelets that operate on S tiles.

Table 2. Benchmarked convolutional layers.

		B	C	C'	Image Size	Padding	Kernel Size
VGG	1.2	64	64	64	$\langle 224, 224 \rangle$	$\langle 1, 1 \rangle$	$\langle 3, 3 \rangle$
	2.2	64	128	128	$\langle 112, 112 \rangle$	$\langle 1, 1 \rangle$	$\langle 3, 3 \rangle$
	3.2	64	256	256	$\langle 56, 56 \rangle$	$\langle 1, 1 \rangle$	$\langle 3, 3 \rangle$
	4.2	64	512	512	$\langle 28, 28 \rangle$	$\langle 1, 1 \rangle$	$\langle 3, 3 \rangle$
	5.2	64	512	512	$\langle 14, 14 \rangle$	$\langle 1, 1 \rangle$	$\langle 3, 3 \rangle$
FusionNet	1.2	1	64	64	$\langle 640, 640 \rangle$	$\langle 0, 0 \rangle$	$\langle 3, 3 \rangle$
	2.2	1	128	128	$\langle 320, 320 \rangle$	$\langle 0, 0 \rangle$	$\langle 3, 3 \rangle$
	3.2	1	256	256	$\langle 160, 160 \rangle$	$\langle 0, 0 \rangle$	$\langle 3, 3 \rangle$
	4.2	1	512	512	$\langle 80, 80 \rangle$	$\langle 0, 0 \rangle$	$\langle 3, 3 \rangle$
	5.2	1	1024	1024	$\langle 40, 40 \rangle$	$\langle 0, 0 \rangle$	$\langle 3, 3 \rangle$
C3D	C2a	32	64	128	$\langle 16, 56, 56 \rangle$	$\langle 1, 1, 1 \rangle$	$\langle 3, 3, 3 \rangle$
	C3b	32	256	256	$\langle 8, 28, 28 \rangle$	$\langle 1, 1, 1 \rangle$	$\langle 3, 3, 3 \rangle$
	C4b	32	512	512	$\langle 4, 14, 14 \rangle$	$\langle 1, 1, 1 \rangle$	$\langle 3, 3, 3 \rangle$
3DUNet	1.2	1	32	64	$\langle 114, 130, 130 \rangle$	$\langle 0, 0, 0 \rangle$	$\langle 3, 3, 3 \rangle$
	2.2	1	64	128	$\langle 54, 62, 62 \rangle$	$\langle 0, 0, 0 \rangle$	$\langle 3, 3, 3 \rangle$
	3.2	1	128	256	$\langle 26, 30, 30 \rangle$	$\langle 0, 0, 0 \rangle$	$\langle 3, 3, 3 \rangle$

Efficient fork–join synchronization A single fork–join is used to execute the statically scheduled tasks. To minimize the synchronization time required for the fork–join operations, we have designed a custom *barrier*. The main thread first assigns (a pointer to) a function to each thread, after which all threads pass the barrier. Then, all threads start executing their tasks. After execution, each thread waits again on the barrier for all the other threads to finish. Finally, the main thread then continues the program execution, while the other threads wait on the barrier again for a new task to be available.

Inspired by the SPIRAL [12] project, we have implemented a custom busy-wait barrier using c++11 atomics and memory model. The synchronization using our custom primitive is done in a fraction of cycles compared to Cilk, openMP or pthread barriers.

5 Experiments

This section reports our experimental results. We would like to answer the following questions:

- What is the performance of our implementation, comparing with other state–of–the–art alternatives for manycore CPUs and GPUs?
- What is the performance of the JIT batched matrix multiplication primitives, comparing with optimized libraries?
- What is the precision loss of the Winograd–based convolution for various $F(m, r)$?

To answer these questions, we benchmarked our implementation on an Intel Xeon Phi 7210 processor, which has 64 cores, capable of approximately 4.5 TFLOPS of single precision floating point, and 16 GB of HBW memory with approximate speed of 400 GBytes/s. To compare with the state-of-the-art methods on GPUs, we evaluated several popular methods on an Nvidia Titan X Pascal, capable of approximately 11 TFLOPS for FP32.

5.1 Convolutional Network Performance

We evaluated our implementation on modern 2D and 3D ConvNets for both object detection and segmentation. Detection networks use multiple batches (B) and gradually down–sample the images while increasing the number of channels; segmentation networks use rather large images with batch size of one.

Tbl. 2 shows different ConvNet layers in our experiments. To evaluate 2D object detection, we chose the VGG-A version of OxfordNet [47], which was an ImageNet competition winner [45] and is widely used for speed benchmarking [6]. As a representative 2D ConvNet for segmentation, we chose FusionNet [42] used for biomedical image segmentation. FusionNet is an encoder–decoder type network, a style used by nearly all modern ConvNets for segmentation such as U–Net [44], DeconvNet [41], and SegNet [14]. As a representative 3D ConvNets, we chose C3D, a spatiotemporal feature learning network [39] and 3D U–Net [18] used for segmentation of volumetric 3D images. We benchmarked the most computationally expensive convolutional layers of each network.

We compared our implementation with other available Winograd–based implementations for CPUs, including FALCON [1], LIBXSMM [10], and MKL-DNN [2]. We also compared with direct convolution implementations including MKL-DNN and the one proposed in [58]. LIBXSMM provides three different data layouts: LIBXSMM, Tensorflow and Mixed layout. We benchmarked all three and reported the fastest. For MKL-DNN, we used nChw16c data layout, which is the same as ours.

Additionally, we benchmarked three cuDNN implementations: Winograd–based for 2D, matrix–multiply based for 3D, and FFT based for 3D data.

Fig. 5 shows the execution times of the convolutional layers from Tbl. 2. For our implementation, we show the speeds of various $F(m, r)$. The columns annotated with "FX" assume no kernel transformation, which are suitable for inference–only computation, where the kernel transformations are pre–computed and re–used.

There are two main results. The first is that our implementation significantly outperforms all other CPU implementations. The second is that when comparing with state-of-the-art GPU implementations on Nvidia Titan X Pascal GPU, our implementation on the Xeon Phi processor achieved competitive performance for 2D and much better performance for 3D.

2D networks FALCON's Winograd implementation only supports $F(2\text{x}2, 3\text{x}3)$. Our implementation with the same $F(m, r)$ achieved speedups between 1.08x and 5.39x. With larger $F(m, r)$, our implementation achieved up to 8.33x speedup.

MKL-DNN and LIBXSMM only support $F(4\text{x}4, 3\text{x}3)$. Our implementation with the same $F(m, r)$ achieved speedups ranging from 1.59x to 2.84x over MKL-DNN and from 1.32x to 4.05x over LIBXSMM. With larger $F(m, r)$, our implementation achieved up to 3.34x speedup over MKL-DNN and 5.07x over LIBXSMM.

For some layers, like VGG-1.2 and all 5 layers from FusionNet, the existing Winograd implementations on CPU did not outperform the direct ones.

CuDNN is not open source and does not document the tile size used in the 2D Winograd implementation. Based on the numerical accuracy (Sec 5.3), we can speculate that it uses F(4x4,3x3). It outperformed ours by an average of 1.5x, while running on a GPU that is capable of roughly 2.5x more FLOPS than the KNL processor. Thus we conclude that our implementation better utilizes the hardware.

While there is no publicly available code for the Winograd algorithm proposed by Budden et al. [15] for CPUs, we can make comparisons based on the benchmarks provided in their manuscript. Their measured throughput of the sample network (3 layers with 32 channels each, and unusual kernel size of 4x4) on an 18-core Intel E7-8890 CPU (Haswell) was 10.9 MVox/s. On the same network and CPU, the latest MKL-DNN using direct convolution achieved more than 12 MVox/s. Our approach achieved roughly 100 MVox/s (9x speedup) on the KNL processor. Since the peak FLOPS of the E7-8890 CPU is roughly 1/3 of the KNL processor, we can estimate that our algorithm achieves 3x better utilization of the hardware.

3D networks Since all Winograd (CPU and GPU) implementations support only 2D convolutions, we could only perform limited benchmarks for 3D convolutions. Our implementation achieved better performance than the approach proposed by Zlateski et al. [58] by more than 6x, than cuDNN's matrix–multiply based convolution by more than 2x, and than cuDDN's FFT based convolution by more than 8x.

Inference vs training All speedups reported above are the ones for training – using the implementation that transforms kernels as well. The "FX" variation, that assumes memoized values of the kernel transforms, can further improve the performances in certain cases. For most of the layers, the kernel transformations only require a small percentage of the total execution time. However, for layers with a large number of input/output channels, the kernel transformations can take significant time compared to the time required for transforming the inputs/outputs, especially when

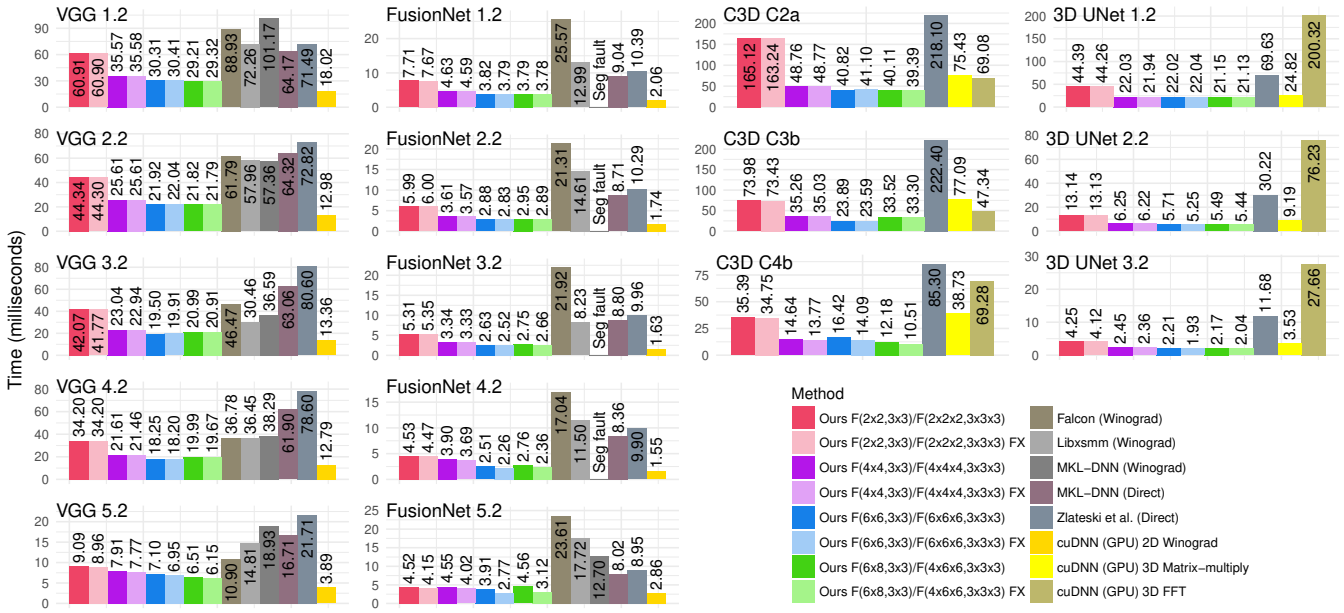

Figure 5. Convolution layers' runtime with different implementations. MKL-DNN's Winograd–based convolution produces segmentation faults for 4 of 5 FusionNet layers. The columns annotated with "FX" do not perform kernel transformations.

the batch size is one. This is notable for FusionNet (layers 4.2 and 5.2).

Effects of $F(m, r)$ Theoretically, the larger the m, the more operations can be saved at the matrix multiplication stage. However, in reality, large values of m can lead to a computation overhead due to the following reasons: (1) The dimension length of output images has to be divisible by m, otherwise, the image is zero padded, increasing the number of operations at both transformation and matrix multiplication stages. This is the main reason why, for some layers, larger ms did not achieve better performance. (2) The number of operations for the image and kernel transformations increases quadratically with m [36].

There is, however, a limit to how big m can be due to precision loss for large m, as analyzed below.

5.2 Batched Matrix Multiplication

We evaluated the performances of our JIT–compiled batch matrix–multiply routines, which were used in the most computationally expensive stage of our algorithm.

We benchmarked batched matrix multiplications against LIBXSMM [27] and Intel MKL [8], which are the most frequently used on x86 CPUs. We measured the throughput (in FLOPS) for relevant matrix sizes. In our experiment, each available core was performing multiplications of tall and skinny matrices $\hat{U}$ with $\hat{V}$ which had less than 128^2 elements, and the overall performance was recorded.

Blocking strategies of computing n_{blk} rows of the resulting matrix at the time were considered ($6 \leqslant n_{blk} \leqslant 30$)

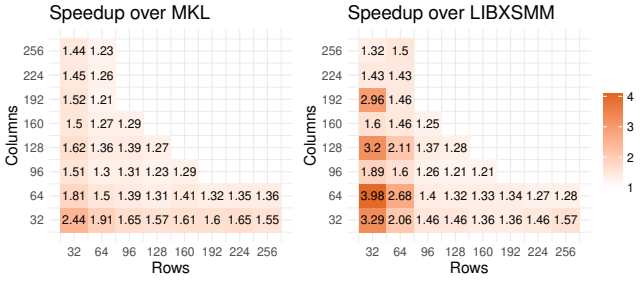

Figure 6. Speedup of our JIT batched matrix multiply primitives over alternatives.

and the fastest one was recorded. LIBXSMM has various prefetching strategies, we reported the one with the highest throughput. We used Intel MKL version 2017.4.196, which has two different interfaces – a BATCH and CBLAS, we benchmarked both and reported the better one.

Fig. 6 shows the speedups of our implementation over MKL and LIBXSMM for various sizes of $\hat{V}$. Overall, our implementation outperformed MKL and LIBXSMM by an average of 60% and 70% respectively. Smaller $\hat{V}$s benefited more from our approach, with speedups up to 2.44x over MKL and 3.98x over LIBXSMM.

Most of the speedup over LIBXSMM comes from our more sophisticated pre–fetching strategies, which is particularly important for small matrix sizes. Additionally, we allow for an arbitrary number of blocking registers (between 6 and 30), and use the optimal number based on empirical results.

Table 3. Element errors of convolution layers.

VGG	Direct	F(2^2,3^2)	F(4^2,3^2)	F(6^2, 3^2)	F(6x8,3^2)	F(8^2,3^2)
Train max	1.11E-06	3.42E-07	7.13E-06	1.30E-03	3.03E-02	8.31E-01
Train avg	3.32E-08	2.17E-08	1.05E-07	4.62E-06	8.10E-05	1.50E-03
Infer max	4.50E-07	2.34E-07	3.85E-06	5.60E-04	1.65E-02	4.50E-01
Infer avg	1.72E-08	1.20E-08	5.67E-08	2.35E-06	4.10E-05	7.30E-04
C3D	Direct	F(2^3,3^3)	F(4^3, 3^3)	F(4x6^2,3^3)	F(6^3, 3^3)	F(8x6^2,3^3)
Train max	1.80E-06	3.90E-07	3.64E-05	4.66E-03	6.69E-02	1.94E+00
Train avg	5.66E-08	2.83E-08	3.06E-07	1.33E-05	9.21E-05	1.71E-03
Infer max	1.82E-06	4.58E-07	2.95E-05	5.36E-03	6.10E-02	1.68E+00
Infer avg	5.54E-08	2.80E-08	3.20E-07	1.37E-05	9.31E-05	1.63E-03

LIBXSMM uses a fixed number of 16 registers, which is not always optimal. A minor improvement comes from allowing for more unrolling. As MKL is not open source, we do not know where most of the speedup comes from.

5.3 Accuracy

We measured the computation errors for various sizes of $F(m, r)$ to determine which sizes are appropriate in practice. As m, in $F(m, r)$ increases, more significant digits are required for accurate computation [34, 55]; the single precision floating point has limited number of bits (24) for the significand, which can greatly increase the computation error [34] for larger ms.

Tbl. 3 shows the maximal and average errors for all layers and various $F(m, r)$, as well as the errors of direct convolution using 32–bit floating points as a reference. The ground truth was estimated using a direct convolution algorithm that uses "long doubles" (extended precision floating point numbers). The input image data is generated from an uniform distribution [-0.1, 0.1].

To measure training errors, we initialized the kernels using Xavier initialization [24]. Inference errors were measured using pre–trained kernels from [5] for VGG and kernels from [4] for C3D.

The precision is more important for training, as it affects the stability, and less for inference. Evidence suggests [19, 25] that errors under E-02 do not affect training stability, whereas the errors for inference can be an order of magnitude higher.

Our measurements show that $F(6^2, 3^2)$ for 2D and $F(4 \times 6^2, 3^3)$ for 3D have errors small enough for training, and that $F(6 \times 8, 3^2)$ for 2D and $F(6^3, 3^3)$ for 3D have errors small enough for inference. However, recent research on low precision ConvNets [52] suggested that slightly larger m might be appropriate in certain application domains.

6 Conclusions

While the Winograd–based convolution algorithm can greatly reduce the computational complexity of convolutional layers, it is challenging to implement it to fully exploit the hardware performance of manycore CPUs.

Our implementation supports N–dimensional ConvNets with arbitrary kernel and transformation sizes, and is substantially faster than the existing Winograd implementations for the CPU, competitive with GPU implementations for 2D and faster for 3D. This was achieved by proposing a series of interdependent optimizations, each motivated by a specific hardware feature or limitation, but designed to work in together to improve the overall performances. As our method was designed from scratch, and not as a series of incremental optimizations of an existing method, it is somewhat difficult to perfectly isolate the contribution of each individual optimization.

The custom data layout facilitated the vectorization of transformation codelets and matrix multiplication routines, as well as minimized TLB misses by organizing memory access patterns properly. Interleaving memory operations with computation amortized the cost of memory access by keeping a high compute–to–memory ratio. On time prefetching ensured that most data is in cache before performing computation. Non–temporal stores avoided cache pollution and consumed less bandwidth, improving the run–times of the transform stages by an average of 25%.

The custom JIT matrix-multiply routines improved the speed of the most computationally expensive stage of the algorithm by an average of 60%, and up to 2.5x; with an additional speedup of 20% to the overall performances due to inclusion of interleaved streaming stores.

The proposed static scheduling algorithm, that evenly divides the work among the cores and minimizes the synchronization overhead with a custom busy–wait barrier, was a major factor of the implementation that enabled a near-ideal speedup.

By analyzing the computation errors, we have demonstrated that the Winograd algorithm is indeed a viable approach to improve the performance of convolution layers.

Finally, we have contributed to the community by releasing our code as an open source project. The current implementation is AVX512 specific. It can be easily extended to support AVX2 instructionset, by providing specific matrix multiplication routines; the rest of the code can be fully reused. The optimization principles are applicable to other SIMT/SIMD architectures, such as GPUs, but would require a separate implementation.

Acknowledgments

We thank Sebastian Seung for helpful discussions. We are grateful to Intel Corporation for supporting the Intel Parallel Computing Center at Princeton University, and to Toyota Research Institute for supporting the Toyota - CSAIL Joint Research Center at MIT. Zhen Jia was partially supported by IARPA (D16PC00005).

References

[1] 2016. FALCON Library: Fast Image Convolution in Neural Networks on Intel Architecture. "https://colfaxresearch.com/falcon-library/". (2016).

[2] 2016. Intel(R) Math Kernel Library for Deep Neural Networks. "https://github.com/01org/mkl-dnn". (2016).

[3] 2018. N-Dimensional Winograd–based convolution framework. https://bitbucket.org/poozh/ond-winograd. (2018).

[4] Accessed: 01-14-2018. C3D: Generic Features for Video Analysis. http://vlg.cs.dartmouth.edu/c3d/. (Accessed: 01-14-2018).

[5] Accessed: 01-14-2018. ILSVRC-2014 model (VGG team) with 16 weight layers. https://gist.github.com/ksimonyan/211839e770f7b538e2d8. (Accessed: 01-14-2018).

[6] Accessed: 01-14-2018. Imagenet Winners Benchmarking. https://github.com/soumith/convnet-benchmarks. (Accessed: 01-14-2018).

[7] Accessed: 01-14-2018. Intel® Intrinsics Guide. https://software.intel.com/sites/landingpage/IntrinsicsGuide/. (Accessed: 01-14-2018).

[8] Accessed: 01-14-2018. Intel® Math Kernel Library. https://software.intel.com/en-us/mkl. (Accessed: 01-14-2018).

[9] Accessed: 01-14-2018. Intel® Nervana reference deep learning framework. https://github.com/NervanaSystems/neon. (Accessed: 01-14-2018).

[10] Accessed: 01-14-2018. LIBXSMM. https://github.com/hfp/libxsmm. (Accessed: 01-14-2018).

[11] Accessed: 01-14-2018. MCDRAM as High-Bandwidth Memory (HBM) in Knights Landing Processors: Developer's Guide. "https://colfaxresearch.com/knl-mcdram/". (Accessed: 01-14-2018).

[12] Accessed: 01-14-2018. SPIRAL Project: Fast x86 Barrier. http://www.spiral.net/software/barrier.html. (Accessed: 01-14-2018).

[13] Accessed: 01-14-2018. Wincnn. "https://github.com/andravin/wincnn". (Accessed: 01-14-2018).

[14] Vijay Badrinarayanan, Alex Kendall, and Roberto Cipolla. 2017. SegNet: A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2017).

[15] David Budden, Alexander Matveev, Shibani Santurkar, Shraman Ray Chaudhuri, and Nir Shavit. 2016. Deep Tensor Convolution on Multicores. *arXiv preprint arXiv:1611.06565* (2016).

[16] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759* (2014).

[17] G. Chrysos. 2012. Intel Xeon Phi X100 Family Coprocessor–The Architecture. white paper, Intel. (Nov 2012).

[18] Özgün Çiçek, Ahmed Abdulkadir, Soeren S Lienkamp, Thomas Brox, and Olaf Ronneberger. 2016. 3D U-Net: learning dense volumetric segmentation from sparse annotation. In *International Conference on Medical Image Computing and Computer-Assisted Intervention*. Springer, 424–432.

[19] Matthieu Courbariaux, Yoshua Bengio, and J David. 2014. Low precision arithmetic for deep learning. *CoRR, abs/1412.7024* 4 (2014).

[20] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*. IEEE, 248–255.

[21] Matteo Frigo and Steven G Johnson. 1998. FFTW: An adaptive software architecture for the FFT. In *Acoustics, Speech and Signal Processing, 1998. Proceedings of the 1998 IEEE International Conference on*, Vol. 3. IEEE, 1381–1384.

[22] Dennis Gannon, William Jalby, and Kyle Gallivan. 1988. Strategies for Cache and Local Memory Management by Global Program Transformation. In *Proceedings of the 1st International Conference on Supercomputing*. Springer-Verlag, London, UK, UK, 229–254. http://dl.acm.org/citation.cfm?id=647970.761024

[23] Liuhao Ge, Hui Liang, Junsong Yuan, and Daniel Thalmann. 2017. 3d convolutional neural networks for efficient and robust hand pose estimation from single depth images. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*.

[24] Xavier Glorot and Yoshua Bengio. 2010. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*. 249–256.

[25] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. 2015. Deep learning with limited numerical precision. In *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)*. 1737–1746.

[26] Lance Hammond, Benedict A. Hubbert, Michael Siu, Manohar K. Prabhu, Michael Chen, and Kunle Olukotun. 2000. The Stanford Hydra CMP. *IEEE Micro* 20, 2 (March 2000), 71–84. https://doi.org/10.1109/40.848474

[27] Alexander Heinecke, Greg Henry, Maxwell Hutchinson, and Hans Pabst. 2016. LIBXSMM: accelerating small matrix multiplications by runtime code generation. In *High Performance Computing, Networking, Storage and Analysis, SC16: International Conference for*. IEEE, 981–991.

[28] Alexander Heinecke, Hans Pabst, and Greg Henry. 2015. Libxsmm: A high performance library for small matrix multiplications. *Poster and Extended Abstract Presented at SC* (2015).

[29] James Jeffers, James Reinders, and Avinash Sodani. 2016. *Intel Xeon Phi Processor High Performance Programming: Knights Landing Edition*. Morgan Kaufmann.

[30] J Jiménez, S Doerr, G Martínez-Rosell, AS Rose, and G De Fabritiis. 2017. DeepSite: Protein binding site predictor using 3D-convolutional neural networks. *Bioinformatics* (2017).

[31] Tamara G Kolda and Brett W Bader. 2009. Tensor decompositions and applications. *SIAM review* 51, 3 (2009), 455–500.

[32] Poonacha Kongetira, Kathirgamar Aingaran, and Kunle Olukotun. 2005. Niagara: A 32-Way Multithreaded Sparc Processor. *IEEE Micro* 25, 2 (March 2005), 21–29. https://doi.org/10.1109/MM.2005.35

[33] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*. 1097–1105.

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

[35] Jiajia Li, Casey Battaglino, Ioakeim Perros, Jimeng Sun, and Richard Vuduc. 2015. An input-adaptive and in-place approach to dense tensor-times-matrix multiply. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 76.

[36] Vijay K. Madisetti. 2009. *The Digital Signal Processing Handbook, Second Edition*. CRC Press.

[37] Michael Mathieu, Mikael Henaff, and Yann LeCun. 2013. Fast training of convolutional networks through ffts. *arXiv preprint arXiv:1312.5851* (2013).

[38] Daniel Maturana and Sebastian Scherer. 2015. 3d convolutional neural networks for landing zone detection from lidar. In *Robotics and Automation (ICRA), 2015 IEEE International Conference on*. IEEE, 3471–3478.

[39] Daniel Maturana and Sebastian Scherer. 2015. Voxnet: A 3d convolutional neural network for real-time object recognition. In *Intelligent Robots and Systems (IROS), 2015 IEEE/RSJ International Conference on*. IEEE, 922–928.

[40] Guido F Montufar, Razvan Pascanu, Kyunghyun Cho, and Yoshua Bengio. 2014. On the number of linear regions of deep neural networks. In *Advances in neural information processing systems*. 2924–2932.

[41] Hyeonwoo Noh, Seunghoon Hong, and Bohyung Han. 2015. Learning Deconvolution Network for Semantic Segmentation. In *Computer Vision (ICCV), 2015 IEEE International Conference on*.

[42] Tran Minh Quan, David GC Hilderbrand, and Won-Ki Jeong. 2016. FusionNet: A deep fully residual convolutional neural network for image segmentation in connectomics. *arXiv preprint arXiv:1612.05360* (2016).

[43] Lawrence R Rabiner and Bernard Gold. 1975. Theory and application of digital signal processing. *Englewood Cliffs, NJ, Prentice-Hall, Inc., 1975. 777 p.* (1975).

[44] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. 2015. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical Image Computing and Computer-Assisted Intervention*. Springer, 234–241.

[45] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. 2015. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision* 115, 3 (2015), 211–252.

[46] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. 2008. Larrabee: A Many-core x86 Architecture for Visual Computing. In *ACM SIGGRAPH 2008 Papers (SIGGRAPH '08)*. ACM, New York, NY, USA, Article 18, 15 pages. https://doi.org/10.1145/1399504.1360617

[47] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[48] Avinash Sodani, Roger Gramunt, Jesus Corbal, Ho-Seop Kim, Krishna Vinod, Sundaram Chinthamani, Steven Hutsell, Rajat Agarwal, and Yen-Chen Liu. 2016. Knights landing: Second-generation intel xeon phi product. *Ieee micro* 36, 2 (2016), 34–46.

[49] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 1–9.

[50] Vincent Vanhoucke, Andrew Senior, and Mark Z Mao. 2011. Improving the speed of neural networks on CPUs. In *Proc. Deep Learning and Unsupervised Feature Learning NIPS Workshop*, Vol. 1. 4.

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

[52] Ganesh Venkatesh, Eriko Nurvitadhi, and Debbie Marr. 2017. Accelerating Deep Convolutional Networks using low-precision and sparsity. In *Acoustics, Speech and Signal Processing (ICASSP), 2017 IEEE International Conference on*. IEEE, 2861–2865.

[53] Kevin Vincent, Kevin Stephano, Michael Frumkin, Boris Ginsburg, and Julien Demouth. 2017. On Improving the Numerical Stability of Winograd Convolutions. (2017).

[54] Yida Wang, Michael J Anderson, Jonathan D Cohen, Alexander Heinecke, Kai Li, Nadathur Satish, Narayanan Sundaram, Nicholas B Turk-Browne, and Theodore L Willke. 2015. Full correlation matrix analysis of fMRI data on Intel® Xeon Phi coprocessors. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 23.

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

[56] Aleksandar Zlateski, Kisuk Lee, and H Sebastian Seung. 2016. ZNN–A Fast and Scalable Algorithm for Training 3D Convolutional Networks on Multi-core and Many-Core Shared Memory Machines. In *Parallel and Distributed Processing Symposium, 2016 IEEE International*. IEEE, 801–811.

[57] Aleksandar Zlateski, Kisuk Lee, and H Sebastian Seung. 2016. ZNNi: maximizing the inference throughput of 3D convolutional networks on CPUs and GPUs. In *High Performance Computing, Networking, Storage and Analysis, SC16: International Conference for*. IEEE, 854–865.

[58] Aleksandar Zlateski and H Sebastian Seung. 2017. Compile-time optimized and statically scheduled ND convnet primitives for multi-core and many-core (Xeon Phi) CPUs. In *Proceedings of the International Conference on Supercomputing*. ACM, 8.

A Artifact appendix

A.1 Abstract

Our artifact provides source code (in C++) for our evaluated benchmarks, along with scripts to compile and run the code in order to regenerate the data for the graphs and tables in the paper. To reproduce the running time benchmarks (Fig. 5), the code should be compiled for and run on an Intel Xeon Phi 7210 manycore processor. The running times for each layer presented in the paper were benchmarked on a desktop system with (Xeon Phi 7210 with microcode 0xffff01a0), and replicated, within a few percentage points, on a server version of the system (microcode 0x19b).

The provided source code allows for performing additional benchmarks, of an arbitrary convolutional layer, on an arbitrary processor that supports the AVX–512 instruction set.

The numerical accuracy experiments can be replicated with any AVX–512 capable CPU, and are expected to be similar to the ones presented in the paper.

A.2 Artifact check-list (meta-information)

- **Algorithm:** Winograd–based Convolutional layers.
- **Program:** Winograd–based Convolutional layers optimized for manycores.
- **Compilation:** GCC version 7.2 or later, binutils version 2.9 or later.
- **Data set:** Pre–trained kernels for the VGG and the C3D ConvNet.
- **Run-time environment:** Linux manjaro 4.
- **Hardware:** Intel Xeon Phi 7210.
- **Run-time state:** Disabled Speed–Step and Turbo Boost in BIOS. MCDRAM in flat mode.
- **Execution:** Running bash scripts as a non–privileged user.
- **Output:** CSV files are generated with run–times for benchmarked layers. R script is provided for regenerating the Fig. 5. Accuracy results are given as an ASCII table.
- **Experiments:** git clone; execute the corresponding scripts; observe the experiment results
- **Workflow frameworks used?:** No
- **Publicly available?:** Yes
- **Artifacts publicly available?:** Yes
- **Artifacts functional?:** Yes
- **Artifacts reusable?:** Yes
- **Results validated?:** Yes

A.3 Description

A.3.1 How delivered

Source code, scripts, and pre–trained kernels are uploaded to the bitbucket repository:

```
https://bitbucket.org/poozh/ond-winograd
```

A.3.2 Hardware dependencies

To reproduce the running time results presented in the paper, Intel Xeon Phi 7210 is required. Benchmarks on any CPU supporting AVX–512 are also possible.

A.3.3 Software dependencies

The "numactl" tool is required for binding the process to use the MCDRAM. Our implementation is linux specific; however, no specific version of the kernel is required. In our implementation we use C++17 (C++1z) features, thus a compiler that supports such features is required (we used GCC, but expect ICC and Clang to work as well).

A.3.4 Data sets

Pre–trained kernels of the VGG and the C3D, extracted from the corresponding caffe models are included in the repository. The original data was obtained from [5] for VGG and from [4] for C3D; it was then converted into a text representation with the memory layout specific to our method.

A.4 Installation

Installation of *numactl* tool is required. On Arch based distributions, such as manjaro, it can be done via:

```
$ sudo pacman -S numactl
```

git might be required for easy cloning of our repository.

R language is optional for visualizing the performance results. The *ggplot2* and *gridExtra* R libraries are required in order to recreate the figure, and can be installed from R shell via:

```
> install.packages("ggplot2")
> install.packages("gridExtra")
```

A.5 Experiment workflow

We suggest that the experiments are performed after a clean restart of the target machine. For accurate results no other processes should run in the background.

Two sets of experiments are provided, and can be executed through bash scripts. We will refer to the root directory of the cloned repository as $*home_win*.

A.5.1 Convolutional Layer Performance

The first set of experiments measures the runtime of our implementation for various ConvNet layers and sizes of $F(m, r)$. The goal is to reproduce the numbers provided on Fig. 5, which were benchmarked on the Xeon Phi 7210 processor. The optimal execution parameters, as described in Sec 4.3.2, threads per core, and row and column blocks for matrix multiplication were empirically determined for the Xeon Phi 7210 processor, thus only measurements for such optimal configurations are performed. The benchmarks can be performed via:

```
$ cd $home_win/
$ ./bench_xeon_7210_specific.sh
```

For each benchmarked layer, a single C++ file is compiled and then executed. In our implementation we extensively use C++ templates in order to move the computation to compile–time as much as possible. For that reason, compilation time is relatively large. The benchmarks should complete in less than 30 minutes.

The logs are shown on the standard output, as well as stored inside *logs* sub–directory. A *measurements.csv* file is also created inside *R* sub–directory, that can be used to recreate the Fig. 5 using the provided *R* script:

```
$ cd $home_win/R
$ Rscript plot.R
```

For benchmarks on CPUs other than Xeon Phi 7210, which support AVX–512 instruction set, an exhaustive set of tests can be performed via:

```
$ cd $home_win/
$ ./bench_exhaustive.sh $CORES $MEMORY
```

Here, the argument $CORES should be set to the number of available physical cores of the CPU, and $MEMORY should be set to the node representing the MCDRAM, if present (typically 1). When MCDRAM is not present, $MEMORY should be set to 0. The current method uses a single memory node, for that reason, it is expected to under–perform on a multi–socket system. Future work might include additional optimization for such systems.

The exhaustive benchmarks will attempt using various configurations for the number of threads per core, row and column blocks for matrix multiplication in order to empirically find the fastest one. For that reason, the exhaustive benchmarks will take significantly more time than the ones pre–tuned for the Xeon Phi 7210. The runtime will depend on the target CPU, and can take up to an hour per benchmarked layer.

Similarly, the *measurements.csv* file will be created, from which a figure can be generated. Logs will be displayed on the standard output and stored in the *logs* sub–directory.

A.5.2 Accuracy

For reproducing the accuracy measurements (Tbl. 3), the following script should be executed:

```
$ cd $home_win/
$ ./measure_accuracy.sh $CORES
```

The results will be displayed on the standard output in a form of an ASCII table.

A.6 Evaluation and expected result

The generated figure, which visualizes the results and is named *benchs.pdf*, can be found in the *R* sub–directory. A CSV format result set, i.e., *measurements.csv*, can also be examined if the system does not support *R* language.

The benchmarks of the convolutional layer's running times that are specific for the Xeon Phi 7210 should be similar to the ones provided on Fig.5.

The benchmarks on other CPUs should have running times roughly proportional to the computational capability (FLOPS) of the target system and its memory throughput.

The results of the measured numerical accuracies should be similar to the ones given on Tbl. 3.

A.7 Experiment customization

Arbitrary convolutional layers can be benchmarked, both exhaustevely, and with specific settings through the calls of templated C++ functions *do_bench* and *do_bench_specific*. Example usages can be found in the *bench_win/example.cpp* and *bench_win_specific/example.cpp*.

A.8 Notes

The authors also provide the code used to benchmark the competing implementation in the *competing* sub–directory of the repository. We encourage the reviewers to follow the installation instruction for each individual competitor.

Provided results were obtained in late August 2017, and replicated in late October 2017.

The FALCON library did not have any changes since, while the MKL–DNN and LIBXSMM might have some, as they are actively under development.

Our provided implementation can help the reviewer benchmark LIBXSMM and MKL–DNN, as their documentation for the Winograd–based implementation is very scarce. Since they are under active development, their APIs and interfaces are subject to change as well.