2022 IEEE 34th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD) | 978-1-6654-5155-0/22/$31.00 ©2022 IEEE | DOI: 10.1109/SBAC-PAD55451.2022.00027

# Convolution Operators for Deep Learning Inference on the Fujitsu A64FX Processor

Manuel F. Dolz
*Universitat Jaume I de Castellón, Spain*
dolzm@icc.uji.es

Héctor Martínez
*Universidad de Córdoba, Spain*
el2mapeh@uco.es

Pedro Alonso, Enrique S. Quintana-Ortí
*Universitat Politècnica de València, Spain*
palonso@dsic.upv.es,quintana@disca.upv.es

***Abstract*—The convolution operator is a crucial kernel for many computer vision and signal processing applications that rely on deep learning (DL) technologies. As such, the efficient implementation of this operator has received considerable attention in the past few years for a fair range of processor architectures.**

**In this paper, we follow the technology trend toward integrating long SIMD (single instruction, multiple data) arithmetic units into high performance multicore processors to analyse the benefits of this type of hardware acceleration for latency-constrained DL workloads. For this purpose, we implement and optimise for the Fujitsu processor A64FX, three distinct methods for the calculation of the convolution, namely, the lowering approach, a blocked variant of the direct convolution algorithm, and the Winograd minimal filtering algorithm. Our experimental results include an extensive evaluation of the parallel scalability of these three methods and a comparison of their global performance using three popular DL models and a representative dataset.**

***Index Terms*—Convolutional neural networks, high performance, SIMD arithmetic units, ARM-based A64FX processor.**

## I. Introduction

Many current multicore architectures integrate several levels of cache to hide the memory access cost as well as SIMD (single instruction, multiple data) arithmetic units to improve raw performance by operating simultaneously with multiple data items grouped as a vector [1]. Concerning the latter, the trend in past decades has been to expand the width of these units, for example, from 64 bits in MMX (Intel Pentium MMX, AMD K6-2; 1997–1998), to 128 bits in SSE (Intel Pentium III, AMD Athlon XP; 1999), 256 bits in AVX (Intel Sandy Bridge, AMD Bulldozer; 2011), and 512-bit in AVX-512 (Intel Xeon Phi, Intel Skylake; 2013). In addition, SIMD units are also present in many processors of IBM's PowerPC series; ARM's scalable vector extension (SVE) supports SIMD units of up to 2048 bits; and SIMD extensions have been proposed for the RISC-V instruction set architecture (ISA).

While the SIMD-oriented evolution of the hardware is intended to boost performance, whether this improvement can be realised in practice depends on the target algorithm and operand dimensions. Along this line, in this work, we investigate the benefits and caveats of exploiting long, 512-bit SIMD units for deep learning (DL) inference workloads with latency constraints. In particular, we address the high-performance calculation of the convolution operator, a key kernel for many DL tasks in computer vision and signal processing [2], [3], using three distinct methods on the Fujitsu A64FX processor. In doing so, we make the following specific contributions:

- For the lowering approach [4]–[6], we integrate an implementation of the IM2ROW transform with a high-performance A64FX-specific instance of the general matrix multiplication (GEMM) based on the basic linear algebra instantiation framework (BLIS) [7].
- We develop BLIS-like A64FX-specific microkernels which are then leveraged to obtain an efficient blocked variant of the direct convolution algorithm [3], [8].
- We modify the Winograd minimal filtering algorithm significantly to adapt it to the 512-bit SIMD units of the A64FX [9], [10].
- Using three popular DL models and a reference dataset, we conduct a complete evaluation of the previous three methods on the A64FX, focused on the impact of the SIMD units and the multi-threaded parallelisation.

In summary, we explore the efficiency of the 512-bit SIMD vectorisation and OpenMP-based parallelisation of three popular methods for the convolution, on the Fujitsu A64FX processor, for latency-constrained DL workloads.

The rest of the paper is structured as follows. In Section II we briefly review the convolution operator together with a basic realisation. In the following three sections, we then present the A64FX implementations for the three target methods: the lowering approach in Section III; the direct convolution in Section IV; and the Winograd algorithm in Section V. Next, in Section VI, we report the results of our experimental evaluation. Section VII reviews some related libraries that support the convolution algorithms presented in this work. Finally, the paper is closed in Section VIII with a few concluding remarks.

## II. Brief Introduction to the Convolution

The convolution operator

$$O = \textsc{conv}(F, I), \qquad (1)$$

receives 4-dimensional (4D) input and filter tensors, respectively $I$ and $F$, to produce a 4D output tensor $O$, where:

- $I$ consists of $b$ input images of size $h_i \times w_i \times c_i$ each, $h_i \times w_i$ denote the image height $\times$ width, and $c_i$ stands for the number of input channels.



Work licensed under Creative Commons Attribution NonCommercial, No Derivatives 4.0 License. https://creativecommons.org/licenses/by-nc-nd/4.0/

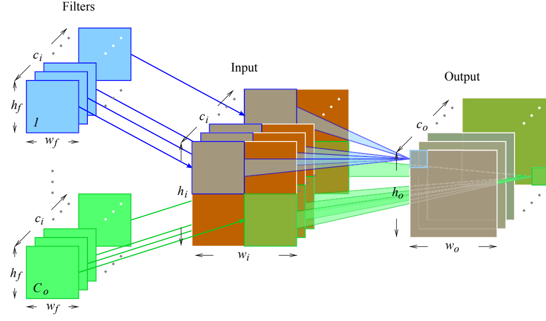


Fig. 1: Sliding window of $c_o$ filters over a single input image producing the output of the convolution operator for each output channel.

- Similarly, $O$ is composed of $b$ outputs of size $h_o \times w_o \times c_o$ each, $h_o \times w_o$ represent the output height $\times$ width, and $C_o$ is the number of output channels.
- Finally, $F$ comprises $c_o$ filters of dimension $h_f \times w_f \times c_i$ each, and $h_f \times w_f$ correspond to the filter height $\times$ width.

For simplicity, we assume hereafter that the filter is applied with unit vertical/horizontal strides, and the output is not padded. In consequence, $h_o = h_i - h_f + 1, w_o = w_i - w_f + 1$.

```
1  void ConvDirect( I[b][h_i][w_i][c_i],
2                   F[c_i][h_f][w_f][c_o],
3                   O[b][h_o][w_o][c_o] ) {
4  for ( h = 0; h < b; h++ )
5    for ( i = 0; i < c_i; i++ )
6      for ( j = 0; j < c_o; j++ )
7        for ( k = 0; k < w_o; k++ )
8          for ( l = 0; l < h_o; l++ )
9            for ( m = 0; m < w_f; m++ )
10             for ( n = 0; n < h_f; n++ )
11               O[h][l][k][j]  +=  I[h][l + n][k + m][i]
12                               ·  F[i][n][m][j];
13 }
```

Listing 1: Basic algorithm for the direct convolution.

The basic algorithm for the direct convolution in Listing 1 shows that each filter convolves a subtensor of the inputs, with the same dimension as the filter, to render a single scalar value (entry) for one of the $c_o$ outputs. The filter is then repeatedly applied to the whole input, in a sliding window manner, to produce the complete entries of this single output; see Figure 1 and [3].

The basic algorithm for the direct convolution consists of 7 nested loops traversing the $(b, c_i, c_o, w_o, h_o, w_f, h_f)$ dimensions of the problem. The ordering of the loops, together with the layout of the tensors, dictate the memory access pattern. The algorithm there adopts the standard NHWC format for the input/output tensors and the CRSK format for the filter tensor. (A popular alternative is to store the input/output tensors in the NCHW format and the filter tensor in the KCRS format.) Furthermore, as the loops in the algorithm are independent of each other, they can be reorganised in any other order. At this point, we note that, for latency-constrained scenarios, the input images have to be processed as soon as they arrive, which implies that $b = 1$. This corresponds to the so-called single-stream case of the ML Commons benchmark for inference on the edge (see https://mlcommons.org/).

In the following sections, we review three different methods to compute the convolution operator and their high-performance implementations on the A64FX, exploiting the 512-bit SIMD units in this processor via ARM SVE intrinsics. In particular, we target 1) the lowering approach; 2) a blocked variant of the direct algorithm; and 3) the Winograd algorithm. The general view of these methods and their high-performance implementations is that the best option largely depends on the parameters that define the convolution.

## III. The Lowering Approach

A high-performance realisation of the convolution operator can be obtained for current computer architectures by casting this operator in terms of a large matrix-matrix multiplication (GEMM). For this purpose, assuming the input/output tensors are stored following the NHWC layout (and the filters in the CRSK layout), the lowering approach:

1) Initially applies the IM2ROW transform [4] to the 4D input tensor $I$ in order to build an augmented 2D matrix $A$, of size $m \times k = (bh_ow_o) \times (c_ih_fw_f)$, as shown in the algorithm in Listing 2.
2) Computes the output of the convolution directly from the GEMM:
$$C = A \cdot B, \tag{2}$$
where $C \equiv O$ is the output tensor, viewed as an $m \times n = (bh_ow_o) \times c_o$ matrix; and $B \equiv F$ is the filter tensor, viewed as a $k \times n = (c_ih_fw_f) \times c_o$ matrix.

These two steps are combined graphically in Figure 2. The lowering approach performs exactly the same arithmetic operations as the direct convolution in Listing 1, and therefore, has the same numerical properties.

```
1  void Im2Row( A[bh_o w_o][c_i h_f w_f], I[b][h_i][w_i][c_i] )
2  {
3  for ( h = 0; h < b; h++ )
4    for ( i = 0; i < c_i; i++ )
5      for ( k = 0; k < w_o; k++ )
6        for ( l = 0; l < h_o; l++ ) {
7          r = l + kh_i + hw_i h_i;
8          for ( m = 0; m < w_f; m++ )
9            for ( n = 0; n < h_f; n++ ) {
10             c = n + mh_f + iw_f h_f;
11             A[r][c] = I[h][l + n][k + m][i];
12 }      }     }
```

Listing 2: Algorithm for the IM2ROW transform.

### A. *High performance* GEMM *and lowering*

On the positive side, the large dimension of the GEMM appearing in the lowering approach favours an efficient exploitation of long SIMD units and exposes a high degree of loop-level parallelism for multicore architectures. In addition, the lowering approach simply needs to invoke an existing high-performance implementation of GEMM, such as that in the BLIS framework [7] for the A64FX [11], to obtain a seamless vectorisation/parallelisation.

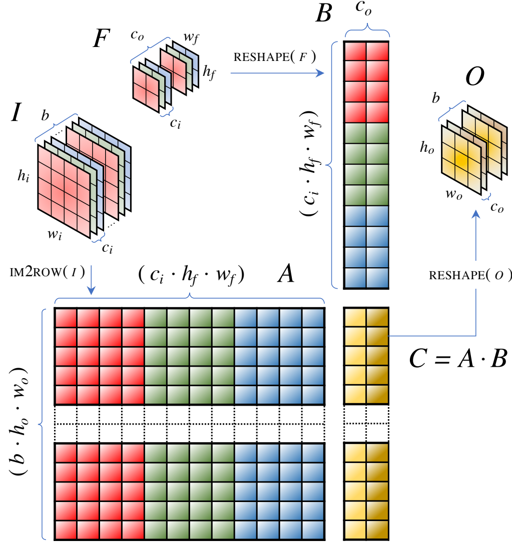


Fig. 2: Convolution operator via the IM2ROW transform. Check also the order of the dimensions for $A$ and $B$. For example, $bh_ow_o$ instead of $h_ow_ob$.

### B. *Accelerating the* IM2ROW *transform*

On the negative side, the lowering approach requires a large workspace $A$ ($h_fw_f$ times larger than $I$), and incurs a certain overhead due to the data copies for the IM2ROW transform.

Exploiting SIMD parallelism in the IM2ROW transform is not straightforward. This kernel only performs data movements between $I$ and the workspace $A$, so we can only benefit from SIMD loads/stores. Let us inspect in some detail the algorithm in Listing 2 though. Assuming a row-major storage for the tensors, and reordering the algorithm so that the loop indexed by $i$ (traversing the $c_i$ dimension of the problem) becomes the innermost loop, we can observe that the algorithm reads $c_i$ contiguous elements of $I$ (innermost loop indexed by $i$) to then copy these entries into non-contiguous positions of $A$. Therefore, we can only leverage SIMD loads to retrieve the data of $A$ into the processor SIMD registers, but we cannot then employ SIMD stores to write these values to the corresponding positions in $A$.

In contrast, parallelising the IM2ROW transform can be easily done by adding the appropriate OpenMP directive to the most appropriate loop(s), preferably one of the outermost ones or a collapsed combination of them.

## IV. The Blocked Algorithm for the Direct Convolution

In [12], we combined the blocking strategy in [8] for the direct convolution algorithm with the packing schemes employed in the high-performance formulation of GEMM [7]. The result was a new blocked version of the direct convolution, illustrated by the algorithm in Listing 3, with the following properties:

- Mimicking the high-performance realisations of GEMM in GotoBLAS2, BLIS, OpenBLAS, and AMD AOML, all the arithmetic is enclosed inside a micro-kernel that computes a GEMM to update a small $m_r \times n_r$ micro-tile of the result (in this case, $O$).
- The dimensions of the micro-tile are decoupled from the cache blocking parameters $w_{o,b}, c_{o,b}, c_{i,b}$.
- The contents of the input tensor are packed into an $m_c \times n_c$ buffer $A_c$ to allow that its entries are accessed with unit stride from the micro-kernel. (For simplicity, the algorithm in Listing 3 only indicates the loop inside which the packing routine is placed.)
- The filter tensor is re-packed into a 5D tensor, of dimension $h_f \times w_f \times c_o/c_{o,b} \times c_i \times c_{o,b}$. As the filters do not vary during inference, this only needs to be done once for the DNN model and its cost becomes negligible. This type of packing enables unit-stride accesses to $B$ from the micro-kernel.

A significant key to attaining high performance in the blocked direct convolution lies in the utilisation of an architecture-specific micro-kernel. Contrary to [8], the decoupling of the micro-tile dimensions from the cache blocking parameters in our case, combined with the packing of the input tensor [12], permits leveraging existing high-performance micro-kernels, specifically tuned for a concrete processor architecture.

```
1  void ConvDirect_Blocked( I[b][h_i][w_i][c_i],  F[c_i][h_f][w_f][c_o],
2                           O[b][h_o][w_o][c_o] ) {
3  for ( h = 0; h < b; h++ )
4   for ( i' = 0; i' < c_i/c_{i,b}; i'++ )
5    for ( l = 0; l < h_o; l++ )
6     for ( k' = 0; k' < w_o/w_{o,b}; k'++ )
7      for ( n = 0; n < h_f; n++ )
8       for ( m = 0; m < w_f; m++ ) {
9        // Here packing for A_c
10       // Ommitted for simplicity
11       for ( j' = 0; j' < c_o/c_{o,b}; j'++ )
12        for ( jj = 0; jj < c_{o,b}; jj += n_r )
13         for ( kk = 0; kk < w_{o,b}; kk += m_r )  {
14          // Micro-kernel
15          for ( ii = 0; ii < c_{i,b}; ii++ )
16           for ( j_r = kk; j_r < kk + m_r; j_r++ )
17            for ( i_r = c_{o,b}; i_r < c_{o,b} + n_r; i_r++ )
18             O[h][l][k' · w_{o,b} + j_r][j' · c_{o,b} + i_r]
19              += I[h][l + n][k' · w_{o,b} + j_r + m][i' · c_{i,b} + ii]
20               ·  F[j' · c_{o,b} + i_r][n][m][i' · c_{i,b} + ii];
21 }     }  }
```

Listing 3: Blocked variant of the direct convolution.

### A. *Micro-kernels for high performance*

The BLIS framework [7] favours portability by encoding most of the GEMM kernel in a high-level programming language such as C. An efficient exploitation of the cache hierarchy is then attained via a proper configuration of a few blocking parameters [13], and the only architecture-specific piece of code boils down to the micro-kernel, which is usually encoded in a low-level programming language.

The BLIS micro-kernel computes the GEMM

$$C_r = \alpha \, A_r \cdot B_r + \beta \, C_r, \tag{3}$$

where $\alpha, \beta$ are both scalars, $C_r$ is a micro-tile of dimension $m_r \times n_r$, and $A_r$ and $B_r$ are micro-panels of dimensions

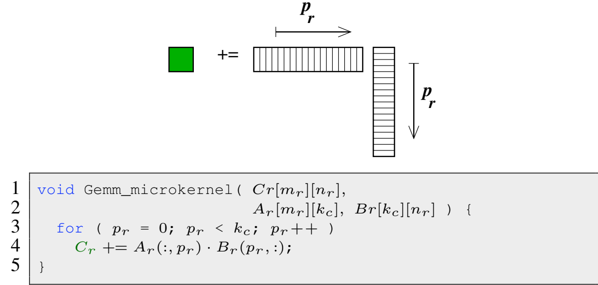


Fig. 3: The BLIS micro-kernel.

respectively given by $m_r \times k_c$ and $k_c \times n_r$. (For the blocked direct convolution, $C_r$ is part of the output tensor, $A_r$ of the augmented matrix, $B_r$ of the filter tensor, and $k_c = c_{i,b}$.) Furthermore, $A_r, B_r$ are stored contiguously in memory, respectively in column- and row-major order, as a result of previous packings, while $C_r$ can be stored in either column- or row-major order, but not necessarily contiguously [7]. Hereafter we will consider the case $\alpha = 1,\ \beta = 1$, which is the one necessary in the blocked direct convolution for DL inference.

Most high-performance implementations of the BLIS micro-kernel, (including that specifically developed for the A64FX,) materialise the micro-kernel as a sequence of $k_c$ outer products, each updating the full $m_r \times n_r$ micro-tile $C_r$; see Figure 3. The outer products are then decomposed into a number of AXPY (scalar $\alpha$ times $x$ plus $y$) vector operations, each updating (part of) either a column or a row of $C_r$ (depending on whether the entries of the micro-tile are respectively stored column-wise or row-wise). The micro-kernel is usually encoded in assembly, in order to take advantage of low-level prefetching instructions, and the AXPY updates are vectorised using SIMD instructions. Furthermore, $m_r, n_r$ are selected to maximise the utilisation of the hardware SIMD registers, and $k_c$ is determined as a function of $n_r$ and some properties of the processor L1 cache, such as capacity and associativity degree [13].

### *B. Micro-kernels for the blocked direct convolution on the A64FX*

The BLIS framework includes a micro-kernel specifically developed and tuned for the A64FX [11] which delivers very high performance for large matrix multiplications involving "squarish" matrix operands. Unfortunately, the dimensions of the input/output/filter tensors for the blocked direct convolution are often highly rectangular, with one dimension much larger than the others. In these circumstances, an alternative micro-kernel, different from that packed with BLIS for the A64FX, may be more efficient.

Developing a high-performance micro-kernel in assembly is a complex, error-prone task due to the low-level, close-to-hardware features of that programming language. To tackle this, in this work we opted for embedding ARM SVE intrinsics into a C-encoded micro-kernel, in order to ease the development, and thus be able to create multiple distinct micro-kernels. After an evaluation of these variants, we selected a micro-kernel for the A64FX with the following characteristics:

1) The basic datatype is IEEE 32-bit floating-point (FP32).
2) The micro-tile dimension is set to $m_r \times n_r = 14 \times 32$. A total of 28 512-bit SIMD registers (viewed as a matrix with 14 rows and 2 columns,) are dedicated to store the entries of the micro-tile $C_r$. As each SIMD register can store up to 16 FP32 numbers, this results in the required $14 \times (2 \cdot 16) = 14 \times 32$ micro-tile.
3) Two SIMD registers are dedicated to store $(2 \cdot 16 =)$ 32 entries of $B_r$.
4) The entries of $A_r$ are retrieved from memory individually, and then immediately broadcast into a SIMD vector register to participate in the corresponding AXPY update.
5) Each iteration of the micro-kernel loop for $k_c$ repeats the following: Upload a row of $B_r$ (with 32 elements) to then update the full $(14 \times 32)$ micro-tile with respect to a column of $A_r$ (with elements loaded one-by-one) and that row of $B_r$.
6) SIMD fused multiply-add operations (FMA) are employed to update the entries of $C_r$.

A simplified implementation of the micro-kernel is shown in Listing 4. The micro-tile `Cr` (for $C_r$) is assumed to be stored in row-major order, with leading dimension (i.e., stride between consecutive elements in the same column) `ldC`. The micro-panels `Ar`, `Br` (for $A_r$, $B_r$) are stored in column- and row-major order respectively, with leading dimensions `mr`, `nr` (for $m_r$, $n_r$).

### *C. OpenMP parallelisation*

The blocked direct convolution presents a considerable number of independent loops, which offer a rich variety of parallelisation opportunities (loop-level parallelism) that can be exploited, for example, via OpenMP. This is further discussed when analysing the parallel scalability of the algorithms in Section VI.

## V. The Winograd Algorithm

The previous sections described how to obtain high performance implementations of the convolution operator, based on the lowering approach and the direct algorithm, on the A64FX. In doing so, they exposed an increasing level of complexity for this task: From simply encoding the IM2ROW transform plus invoking a high-performance instance of GEMM (specifically tuned for this architecture,) for the lowering approach; to developing an architecture-specific micro-kernel and then leveraging it from the blocked direct convolution. In this section, we show that the Winograd algorithm requires a significant step further along this path, asking for an elaborate reformulation of the underlying scheme in order to exploit the long SIMD arithmetic units in the A64FX.

### *A. Algorithm formulation*

The Winograd (minimal filtering) algorithm provides a method to obtain a realisation of the convolution operator with

```
#define Crow(a1, a2) Cr[ (a1)*(ldC)+(a2) ]
#define mr 14
#define nr 32
#define B0 C00n
#define B1 C01n
#define A0 C10n
#define VSET(V, Vvalue)   { V = svdup_n_f32(Vvalue); }
#define VLOAD(V, Vvalue ) { V = svld1_f32(pred16, Vmem); }
#define VMLA(C, A, B)     { C = svmla_f32_z(pred16, C, B, A); }
#define VLMAx2(C0, C1, B0, B1, A0, Avalue) { VSET(A0, Avalue); VMLA(C0, B0, A0); VMLA(C1, B1, A0); }
#define VUPDATE(Cn, C, Cmem) { Cn = VLOAD(Cmem); C = svadd_f32_z(pred16, C, Cn); svst1_f32(pred16, Cmem, C); }

// Registers for micro-tile of C, A, B and final update
svfloat32_t C00, C01, C10, C11, C20, C21, /* Omitted for brevity */ C013, C113,
            C00n, C01n, C10n, C11n;
// Ensure no read/write beyond limits (16 FP32 numbers)
svbool_t pred16 = svwhilelt_b32_u32(0, 16);

// Set micro-tile of C in SIMD registers to zero
VSET(C00, 0);  VSET(C01, 0);
VSET(C10, 0);  VSET(C11, 0);
// Set C20, C21, C30,... to zero omitted for brevity
VSET(C130, 0); VSET(C131, 0);
// Iterate in loop from pr = 0 to Kc-1
for ( int pr=0, baseA=0, baseB=0; pr<Kc; pr++, baseA+=mr, baseB+=nr ) {
  // Load row of B for current iteration
  VLOAD(B0, &Br[baseB]); VLOAD(B1, &Br[baseB + 16]);
  // Update micro-tile
  VMLAx2(C00,  C01,  B0, B1, A0, Ar[baseA + 0]);
  VMLAx2(C10,  C11,  B0, B1, A0, Ar[baseA + 1]);
  // Update of C20, C21, C30, C31,... omitted for brevity
  VMLAx2(C130, C131, B0, B1, A0, Ar[baseA + 13]);
}
// Load contents of C from memory, add to micro-tile,..., and store back in memory
VUPDATE(C00n, C00, &Crow(0, 0)); VUPDATE(C01n, C01, &Crow(0, 16));
VUPDATE(C10n, C10, &Crow(1, 0)); VUPDATE(C11n, C11, &Crow(1, 16));
// Reuse C00n, C01n, C10n, C11n to avoid register spilling
VUPDATE(C00n, C20, &Crow(2, 0)); VUPDATE(C01n, C21, &Crow(2, 16));
VUPDATE(C10n, C30, &Crow(3, 0)); VUPDATE(C11n, C31, &Crow(3, 16));
// Remaining operations, for C40, C41, C50, C51,..., omitted for brevity
```

Listing 4: Structure of the $14 \times 32$ micro-kernel that computes $C_r = A_r \cdot B_r + C_r$, implemented using ARM SVE intrinsics.

a reduced arithmetic cost [14]. Concretely, given a convolution layer that applies a filter $F$ to a single input image $I$, consisting of $c_i$ input channels, in order to produce a single output $O$, with $c_o$ channels, the Winograd-based convolution can be expressed as

$$O_j = M^T \left( \sum_{i=1}^{c_i} \left( G\, F_{j,i}\, G^T \right) \odot \left( H^T\, I_i\, H \right) \right) M, \quad j = 1, 2, \ldots, c_o, \tag{4}$$

where $G$ and $H$ respectively denote the transformation matrices for the filter and input matrices; $M$ is the inverse transformation matrix; $F_{j,i}$ is the $i$-th channel of the $j$-th filter; $I_i$ is the $i$-th channel of the input image; $O_j$ is the $j$-th channel of the output; and $\odot$ denotes the Hadamard (or element-wise) multiplication [9].

From a practical point of view, the 2D Winograd-based convolution applies an $r \times r$ filter to a $t \times t$ input tile in order to produce an $s \times s$ output tile, with $t = s + r - 1$. An input image $I$, of dimension $h_i \times w_i$ (and consisting of $c_i$ input channels), is processed by partitioning it into $t \times t$ tiles, with an overlapping factor of $r - 1$ elements between neighbouring tiles, yielding $\lceil h_i/s \rceil \lceil w_i/s \rceil$ tiles per channel. In this algorithm, choosing a larger value for $s$ reduces the number of arithmetic operations, unfortunately at the cost of introducing numerical instability in the computation [15]. For that reason, $s$ is usually set to be small, with two popular cases being $W(s \times s, r \times r) = W(2 \times 2, 3 \times 3)$ and $W(4 \times 4, 3 \times 3)$.

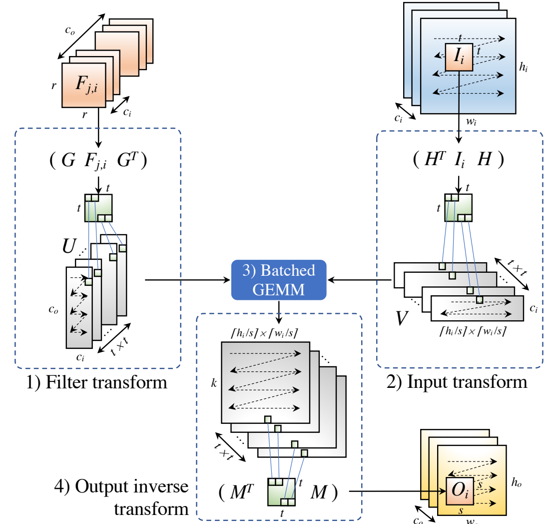


Fig. 4: Workflow of the GEMM-based Winograd algorithm.

According to the Winograd formula in (4), the intermediate Hadamard products are summed over all $c_i$ channels to pro-

duce the $j$-th output channel. Thus, by properly scattering each filter and input transformed tile along the $t \times t$ dimensions on two intermediate workspaces, $U$ and $V$ respectively of sizes $t \times t \times c_o \times c_i$ and $t \times t \times c_i \times (\lceil h_i/s \rceil \lceil w_i/s \rceil)$, both the Hadamard products and the element-wise summations can be collapsed into $t \times t$ independent matrix-matrix multiplications (also known as a "batched" sequence of GEMM). Finally, the same coordinates of the resulting $t \times t$ matrices are gathered to form a new $t \times t$ tile, which is then used to compute the inverse transform as an $s \times s$ tile on the output tensor.

Figure 4 depicts the general workflow of this variant of the Winograd algorithm, exposing the four major phases: 1) filter transform; 2) input transform; 3) "batched" GEMM; and 4) output inverse transform. In the example, the algorithm receives input and filter tensors, respectively of size $h_i \times w_i \times c_i$ and $c_i \times r \times r \times c_o$, to produce an output tensor of size $h_o \times w_o \times c_o = (h_i - r + 1) \times (w_i - r + 1) \times c_o$.

To close this brief review of the Winograd algorithm, in DL applications the aforementioned 3D input/output tensors may comprise an additional dimension, $b$, for the number of independent images to process.

### B. Vectorising the input transform

For the GEMM-based Winograd convolution, vectorising the input transform with different values of $s$ and $r$ entails re-implementing phase 2 in Figure 4. This is due to the distinct dimensions and sparsity patterns of the transformation matrix $H$ generated on the $s + r - 2$ polynomial interpolation points [9]. Given that the most popular alternatives leverage small values of $s$ and $r$, such as $W(2 \times 2, 3 \times 3)$ and $W(4 \times 4, 3 \times 3)$, vectorising this phase on the A64FX requires unrolling (up to a certain degree) the loops iterating over the input tiles in order to fully exploit the 512-bit SIMD units. This technique, referred to as macro-tiling, aims at processing a horizontal (and optionally vertical) block of consecutive tiles of the input images in a single iteration so that the macro-tile columns, stored in vector registers, exploit their full length. Depending on the Winograd variant, the macro-tile can thus accommodate a different number of tiles in both the horizontal and vertical axes.

Figure 5 illustrates the macro-tiling technique for the input transform (phase 2 of the Winograd-based convolution), with $W(2 \times 2, 3 \times 3)$, for simplicity targeting 256-bit SIMD units, able to operate with up to 8 FP32 numbers. In this case, the application of the input transform to the tiles of a macro-tile is split into two sub-operations. The first performs the multiplication $D_i' = H^T \cdot D_i$, where $D_i$ is a macro-tile of size $h_t \times w_t = 6 \times 8$, aggregating a block of $I_i$, comprising $2 \times 3$ input tiles of size $t \times t = 4 \times 4$ overlapping each other $r - 1$ rows and columns. By overlapping $r - 1$ columns between neighbouring tiles, the number of arithmetic operations is reduced by a factor of $1 - \frac{t \times w_t}{t + s \times (w_t - 1)}$, given that these can be leveraged for the tiles that are immediately on the right. Unfortunately, the results related to the $r - 1$ overlapping rows in $D_i$ cannot be leveraged for the tiles that are immediately below. In this case though, the aggregation of two rows of tiles in $D_i$ yields a square $8 \times 8$ matrix that is easy to transpose. The second multiplication $V_i^T = H^T \cdot D_i'^T$ uses the previously transposed macro-tile $D_i^T$ and is computed similarly, with the exception that there are no overlapping columns in the transposed resulting matrix $V_i^T$. This macro-tiling technique can be generalised for any other values of $r, s$, and permits taking advantage of long vector registers. However, an implementation using ARM SVE intrinsics has to be manually customised to operate with the appropriate number of elements (i.e., $hht$ and $wt$), which is contrary to the SVE's focus on code portability.

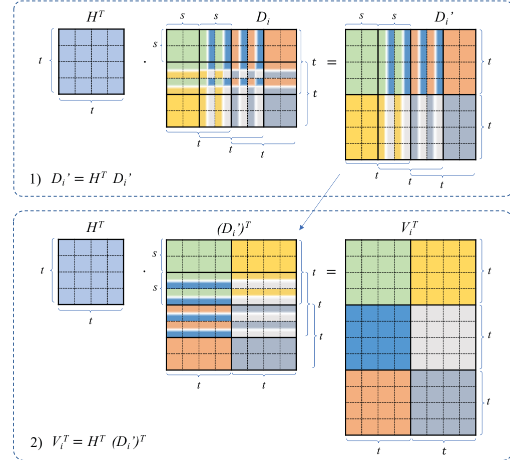


Fig. 5: Example of the Winograd macro-tiling technique for computing the input transform $V_i = H^T I_i H$ using the $W(2 \times 2, 3 \times 3)$ alternative on 256-bit registers with capacity for 8 FP32 numbers.

At this point, we note that the filter transform only needs to be computed once, independently of the number of images to process, and that this can be done offline. Therefore, its contribution to the inference time is negligible. (This is similar to the transform of the filter tensor in the blocked direct convolution.) Also, the output transform was vectorised using the macro-tiling technique applied to the input transform but, due to the more complex access pattern for the result tensor, it did not contribute to any performance improvement.

### C. Exploiting thread-level parallelism using OpenMP

In addition to the introduction of SVE intrinsics, the four phases of the algorithm can be individually parallelised using OpenMP, as the kernels involved by the transform matrices for the filter/input/output tiles present no data dependencies. To augment loop-level parallelism, we used the OpenMP `collapse` clause to fuse the first two loops in each phase: across the $c_o$ and $c_i$ dimensions in phase 1; the $b$ and $c_i$ dimensions in phase 2; the two loops iterating over $t$ in phase 3; and the $b$ and $c_o$ dimensions in phase 4. Each individual $t \times t$ GEMM kernel in phase 3 (see Figure 4) is executed serially but we parallelise their calculation across the $t \times t$ dimensions. Thus, each GEMM kernel in our

Winograd implementation is executed sequentially in order to avoid the exploitation of nested parallelism by setting the `OMP_MAX_ACTIVE_LEVELS` environment variable to `1`.

## VI. Experimental Results

### A. Experimental setup

The following experiments were performed using IEEE FP32 arithmetic on a single node of the MareNostrum 4 CTE-ARM cluster, a platform equipped with Fujitsu FX1000 nodes at the Barcelona Supercomputing Center (BSC). Each node of this cluster features a 48+4-core Fujitsu A64FX processor running at 2.20 GHz, with the cores grouped into 4 Core Memory Groups (CMGs), each with 12 compute cores plus an additional assistant core for the operating system. The cluster nodes are equipped also with 32 GiB of HBM2 memory (per node) and run a Red Hat Enterprise Linux 8 Operating System with the following software used for the experiments: GCC v10.2.0, Python v3.9.13, Numpy v1.23.0rc1, and BLIS v0.8.1. For the experiments, we pinned a single worker thread per physical (compute) core via `numactl`. To ensure reproducibility, all experiments were repeated a large number of times and the results averaged.

During the experiments we measure the execution time of the convolution layers present in three popular DL models: ResNet-50 (v1.5) [3], GoogleLeNet [16], and VGG16 [17], all three combined with the ImageNet dataset [18]. The number of images/batch size was set to $b = 1$ in order to reflect the single-stream scenario of the ML Commons benchmark for inference on the edge, thus prioritising latency over throughput.

### B. Parallel scalability

Our first experiment is designed to expose the parallel scalability of the high-performance implementations of the three target methods on the A64FX. For that purpose, in Figure 6 we report the speedup of each algorithm when running on 4, 8 and 12 (compute) cores of a single CMG (with one thread per core), with respect to its corresponding sequential version. For brevity, we only report the acceleration factors for (the convolution layers of) ResNet-50 but similar results were observed for the two other DNN models. The SIMD-aware A64FX implementation of the Winograd algorithm used the $W(2{\times}2, 3{\times}3)$ variant on a macro-tile of size $h_t{\times}w_t = 10{\times}16$ comprising $4 \times 7$ input tiles. Remind that this algorithm can only be applied to convolution layers with $h_f \times w_f = 3 \times 3$, which explains the smaller number of results in the last plot of the figure.

For the lowering approach (top plot), the results show a stable speedup for the executions on 4 threads, slightly superior to $3\times$ in all except 8 of the last 9 layers. A trend towards a decrease in the speedup is visible for the last layers in the executions with 8 threads and, especially, those involving 12 threads. Concretely, for both thread configurations, we can observe drops in the speedup from layers #41, #83, and #145. For example, for 12 threads, these three drops separate the layers into four groups, with speedups close to $8\times$ up to layer #38 (except for layer #6), in the range $5.9$–$7\times$ for layers #41–#80, around $5\times$ for layers #83–#142, and near to $4\times$ for layers #145–#170 (except for layer #150). This behaviour of the lowering-based algorithm can be related to the dimensions of the GEMM operations present in each of these layer groups.

The blocked direct convolution (middle plot) displays limited parallel scalability, with very low speedups (even smaller than 1, which corresponds to slowdowns) for the initial layers yet slightly growing towards the final layers. For example, for 12 threads the speedup is above $6\times$ for one layer only (#150) and superior to $5\times$ for three other layers (#148, #160, and #170). However, the acceleration factors are much meagre for all other layers. This is motivated by the dimensions of the convolution parameters in ResNet-50, which offer scarce opportunities to extract loop-level parallelism that can be efficiently exploited with OpenMP. Here we tested different options, parallelising distinct loops of the algorithm (see Listing 3) and exploring a task-based parallelisation. We also merged pairs of loops, either automatically using the `collapse` OpenMP clause, or manually by performing a strict distribution of the iteration spaces, with similar or even worse results. In the end, the small values for $c_i$, $c_o$, $h_o$, $w_o$, for many convolution layers in the range of a few dozens, is a strong limiting factor for the scalability of this convolution algorithm.

The speedups of the Winograd algorithm (bottom plot) exhibit a behaviour that is somewhat in the middle between those of the other two algorithms for 4 threads. In contrast, for 12 threads the algorithm exhibits a severe limitation on the parallel scalability, with a speedup of $7\times$ in the first two layers but around or even below $4\times$ in all other cases.

### C. Global comparison

We next conduct a direct comparison of the (high-performance implementations of the three) convolution algorithms. For that purpose, we utilise the GFLOPS (billions of floating-point operations per second) rate, using a flop count of $2c_ic_oh_ow_oh_fw_f$ for all three algorithms. This metric offers a scaled version of the execution time; allows a fair comparison of all three algorithms, independent of the actual number of floating-point operations; and, in contrast with the time, improves the visibility of the results by setting the same upper bound for all layers and models (corresponding to the processor peak performance). To reduce the number of results, we only report the GFLOPS using 12 threads next.

The rates in Figure 7 show that the lowering approach is the global winner in terms of performance due to its superior parallel scalability. (In contrast, when running on a single core, we observed that the blocked direct convolution provides a more efficient solution.) For ResNet-50, the lowering approach delivers between 150 and 250 GFLOPS in most layers while for GoogleLeNet and VGG16 we can observe peaks of up to 600 GFLOPS. In general, the blocked direct convolution is only superior for a few of the final layers in ResNet-50, and Winograd is always outperformed by one of the two other algorithms. (Remind that, for Winograd, results for layers with a filter size different from $3 \times 3$ are not available.)

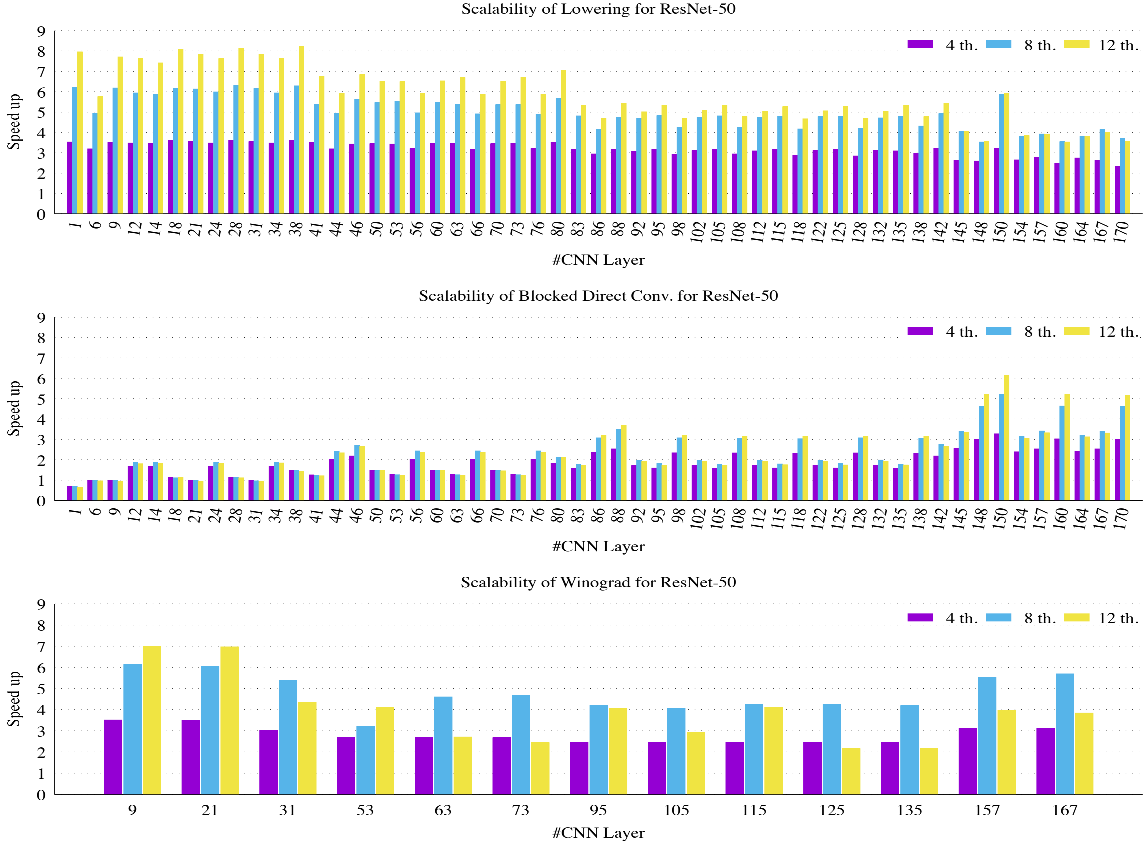


Fig. 6: Speedups of the lowering approach, the blocked direct convolution and the Winograd algorithm (top, middle and bottom, respectively) for ResNet-50+ImageNet using 4, 8 and 12 threads/cores of a single CMG of the A64FX.

### *D. Impact on the DL models*

The three selected DL models comprise a large number of convolution layers but also other types of operators (batch normalisation, non-linear activation functions, fully-connected layers, etc.). Therefore, it is important to put the previous results in perspective by investigating the actual impact of the high-performance implementations of the convolution methods on the total inference time. To analyse this effect, we integrated the distinct convolution options into our PyDTNN framework for high performance inference (and training) DL [19], [20], and performed a complete execution of the inference process.

Figure 8 displays the absolute (aggregated) time for the three DL models. As we have not implemented the Winograd algorithm for convolutional layers with a filter distinct from $3 \times 3$, the convolution operator is computed in those cases using the lowering approach. The consequence is that we can only compare the results for the Winograd algorithm and the lowering approach in the plots of the figure with care; and, more importantly, we should avoid any type of comparison between the Winograd algorithm and the blocked direct convolution, because the former is a mixed option, where the Winograd algorithm is replaced by the highly efficient lowering approach for many layers. The plots in the figure show acceleration factors for the lowering approach over Winograd that range between $1.11\times$ for GoogleLeNet and $1.86\times$ for VGG16; and over the blocked direct convolution between $1.29\times$ for GoogleLeNet and $2.48\times$ for VGG16. This exposes the remarkable advantage of the lowering approach over the two alternatives also for the complete inference process.

## VII. Related work

There exist some vectorised CPU implementations in the literature for the direct, GEMM-based, and Winograd-based convolutions. The implementation of these algorithms for x86-64 CPUs can be found, for instance, in Intel's oneDNN [21] using SIMD AVX2/AVX512 instructions for several data types (INT8, FP32, and BF16). NNPACK [22] also implements these algorithms for x86-64 and ARM architectures using, respectively, AVX2 and NEON intrinsics, though the package has some limitations concerning the filter sizes for the direct and Winograd algorithms. Concretely, for the direct convolution, NNPACK only accepts $1 \times 1$ kernels, while the Winograd implementation is limited to $3 \times 3$ filters, in both

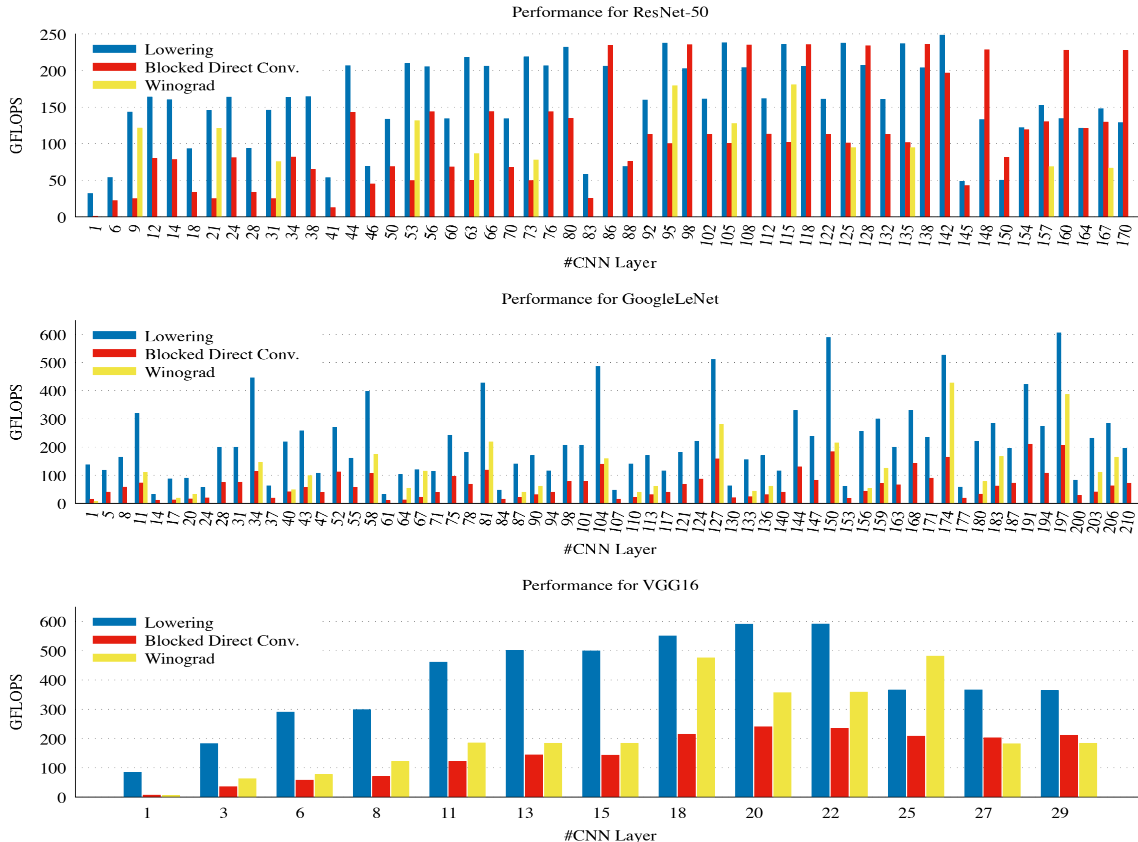


Fig. 7: GFLOPS of the high performance implementations of the convolution algorithms for ResNet-50, GoogleLeNet and VGG16 (top, middle and bottom, respectively) with ImageNet using 12 threads/cores of a single CMG of the A64FX.

cases without any stride. The ARM Compute Library [23] also provides support for these three convolution algorithms with a set of optimisation strategies including SIMD NEON and SVE instructions for different tensor layouts and data types, such as FP32, FP16, INT8, UINT8 and BF16. For the Winograd algorithm, the ARM Compute Library leverages the $W(4\times4, 3\times3)$ variant. However, the implementations that use SVE intrinsics are only available for the GEMM-based convolution. Other libraries, such as FeatherCNN [24], also provide highly-tuned implementations but only for the Winograd-based convolution on ARM CPUs using NEON intrinsics and a combination of algorithmic optimisations that help improve both computational efficiency and memory locality. Although all these libraries have their merits, not all of them include implementations for the three algorithms which exploit the 512-bit SIMD units via SVE intrinsics on the Fujitsu A64FX processor.

## VIII. Concluding Remarks

We have implemented, optimised and evaluated three algorithms for the convolution operator on the Fujitsu processor A64FX, with a strong focus on "long SIMD-vectorisation" and efficient OpenMP-based parallelisation. This effort has produced a number of insights, which are likely to be extensible to other multicore architectures with long SIMD units:

1) In general, the lowering approach can be easily ported to other architectures while retaining much of the performance. The blocked direct convolution can also be efficiently migrated, subject to a manageable architecture-specific optimisation work (especially, when leveraging high-level vector intrinsics instead of assembly code). In contrast, vectorising the Winograd algorithm is a complex task and, even when encoded with SVE, the performance of the resulting codes is not directly portable to architectures with other SIMD lengths.
2) The blocked direct convolution and the Winograd algorithm exhibit limited loop-level parallelism. In consequence, lowering is the best approach in terms of performance when using multiple cores. When running on a single core, the blocked direct convolution provides a competitive alternative and the Winograd algorithm is the best option only in a few cases (due to the difficulties in efficiently exploiting long SIMD units for this algorithm).

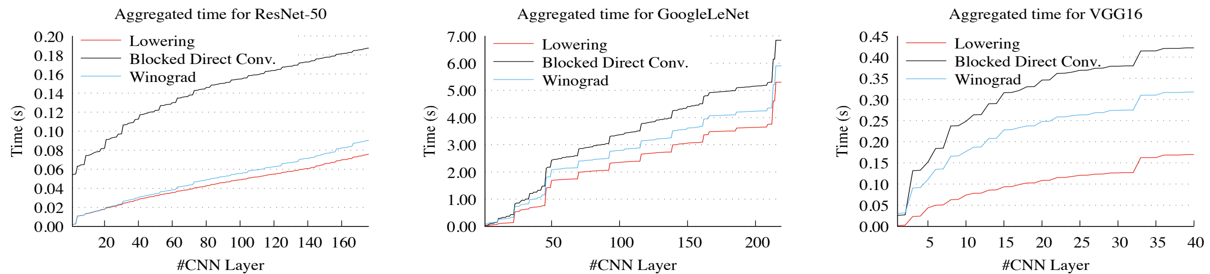


Fig. 8: Aggregated time for the ResNet-50, GoogleLeNet and VGG16 models.

3) The blocked direct convolution requires a special packing of the filters, which in training has to be recalculated in each iteration. A similar comment applies to the calculation of the filter transform for the Winograd algorithm.
4) The lowering approach and the blocked direct convolution are highly flexible, but the former requires a significant additional workspace.

To close this short review, the Winograd algorithm does not have the same numerical properties as the two other methods, although the analysis of that is beyond the scope of this paper. Also, when the goal is DL training, the backward pass for Winograd and the blocked direct convolution are not straightforward to implement.

## Acknowledgments

This research was partially sponsored by projects TIN2017-82972-R of *Ministerio de Ciencia, Innovación y Universidades* and Prometeo/2019/109 of the *Generalitat Valenciana*. Héctor Martínez is a postdoctoral fellow supported by the *Consejería de Transformación Económica, Industria, Conocimiento y Universidades de la Junta de Andalucía*. Manuel F. Dolz was also supported by the Plan GenT project CDEIGENT/2018/014 of the *Generalitat Valenciana*. This project has received funding from the European High-Performance Computing Joint Undertaking (JU) under grant agreement No 955558. The JU receives support from the EU Horizon 2020 research and innovation programme, and Spain, Germany, France, Italy, Poland, Switzerland, Norway.

We thank the Barcelona Supercomputing Center for granting the access to the Fujitsu A64FX system were the developments and tests were performed.

## References

[1] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. San Francisco: Morgan Kaufmann Pub., 2017.

[2] T. Ben-Nun and T. Hoefler, "Demystifying parallel and distributed deep learning: An in-depth concurrency analysis," *ACM Comput. Surv.*, vol. 52, no. 4, pp. 65:1–65:43, Aug. 2019.

[3] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proc. of the IEEE*, vol. 105, no. 12, pp. 2295–2329, 2017.

[4] K. Chellapilla, S. Puri, and P. Simard, "High performance convolutional neural networks for document processing," in *10th Int. Workshop Frontiers in Handwriting Recogn.*, 2006, Université de Rennes, France.

[5] E. Georganas *et al.*, "Anatomy of high-performance deep learning convolutions on SIMD architectures," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, ser. SC '18. IEEE Press, 2018.

[6] P. San Juan, A. Castelló, M. F. Dolz, P. Alonso-Jordá, and E. S. Quintana-Ortí, "High performance and portable convolution operators for multicore processors," in *2020 IEEE 32nd Int. Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 2020, pp. 91–98.

[7] F. G. Van Zee and R. A. van de Geijn, "BLIS: A framework for rapidly instantiating BLAS functionality," *ACM Trans. Math. Softw.*, vol. 41, no. 3, pp. 14:1–14:33, 2015.

[8] J. Zhang, F. Franchetti, and T. M. Low, "High performance zero-memory overhead direct convolutions," in *Proc. 35th Int. Conf. Machine Learning – ICML, Vol. 80, pp. 5776–5785*, 2018.

[9] A. Lavin and S. Gray, "Fast algorithms for convolutional neural networks," in *2016 IEEE Conf. on Computer Vision and Pattern Recogn. (CVPR)*, 2016, pp. 4013–4021.

[10] A. Zlateski, Z. Jia, K. Li, and F. Durand, "The anatomy of efficient FFT and Winograd convolutions on modern CPUs," in *Proc. ACM Int. Conference on Supercomputing*, ser. ICS '19, 2019, p. 414–424.

[11] R. G. Xu, "BLIS & TBLIS on SVE and Apple AMX," 2021, ARM HPC User Group – SC'21. [Online]. Available: https://www.youtube.com/watch?v=xMiWe07Rjss

[12] S. Barrachina *et al.*, "Reformulating the direct convolution for high-performance deep learning inference on ARM processors," *Cluster Computing*, 2022, in review.

[13] T. M. Low, F. D. Igual, T. M. Smith, and E. S. Quintana-Ortí, "Analytical modeling is enough for high-performance BLIS," *ACM Trans. Math. Softw.*, vol. 43, no. 2, pp. 12:1–12:18, Aug. 2016.

[14] S. Winograd, *Arithmetic Complexity of Computations*. Society for Industrial and Applied Mathematics, 1980.

[15] B. Barabasz, A. Anderson, K. M. Soodhalter, and D. Gregg, "Error analysis and improving the accuracy of Winograd convolution for deep neural networks," *ACM Trans. Math. Softw.*, vol. 46, no. 4, Nov. 2020.

[16] C. Szegedy *et al.*, "Going deeper with convolutions," *CoRR*, vol. abs/1409.4842, 2014. [Online]. Available: http://arxiv.org/abs/1409.4842

[17] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[18] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Proc. 25th Int. Conf. Neural Information Processing Systems (NIPS)*, 2012, pp. 1097–1105.

[19] S. Barrachina, A. Castelló, M. Catalan, M. F. Dolz, and J. Mestre, "PyDTNN: a user-friendly and extensible framework for distributed deep learning," *The Journal of Supercomputing*, vol. 77, 09 2021.

[20] S. Barrachina, A. Castelló, M. Catalán, M. F. Dolz, and J. I. Mestre, "A flexible research-oriented framework for distributed training of deep neural networks," in *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2021, pp. 730–739.

[21] Intel, "oneAPI Deep Neural Network Library - oneDNN," 2021. [Online]. Available: https://github.com/oneapi-src/oneDNN

[22] M. Dukhan, "Nnpack: Acceleration package for neural networks on multi-core cpus," 2016. [Online]. Available: https://github.com/Maratyszcza/NNPACK

[23] ARM, "Arm compute library," 2021. [Online]. Available: https://github.com/ARM-software/ComputeLibrary

[24] H. Lan, J. Meng, C. Hundt, B. Schmidt, M. Deng, X. Wang, W. Liu, Y. Qiao, and S. Feng, "FeatherCNN: Fast inference computation with TensorGEMM on ARM architectures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 3, pp. 580–594, 2019.