Journal of Systems Architecture 135 (2023) 102806

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

Reformulating the direct convolution for high-performance deep learning inference on ARM processors

Sergio Barrachina [a], Adrián Castelló [b], Manuel F. Dolz [a,*], Tze Meng Low [c], Héctor Martínez [d], Enrique S. Quintana-Ortí [b], Upasana Sridhar [c], Andrés E. Tomás [a]

[a] Universitat Jaume I, Castellón de la Plana, Spain
[b] Universitat Politècnica de València, Valencia, Spain
[c] Carnegie Mellon University, Pittsburgh, USA
[d] Universidad de Córdoba, Córdoba, Spain

ARTICLE INFO

Keywords:
Convolution
Direct algorithm
Deep learning
High performance
ARMv8 architecture

ABSTRACT

We present two high-performance implementations of the convolution operator via the direct algorithm that outperform the so-called *lowering* approach based on the IM2COL transform plus the GEMM kernel on an ARMv8-based processor. One of our methods presents the additional advantage of zero-memory overhead while the other employs an additional yet rather moderate workspace, substantially smaller than that required by the IM2COL+GEMM solution. In contrast with a previous implementation of a similar zero-memory overhead direct convolution, this work exhibits the key advantage of preserving the conventional NHWC data layout for the input/output activations of the convolution layers.

1. Introduction

The convolution (operator) is a key computational kernel that concentrates a significant part of the arithmetic cost for the type of deep neural networks (DNNs) that are frequently leveraged in signal processing and computer vision tasks [1,2]. Therefore, it is natural that a considerable effort has been dedicated to the efficient implementation of this operator. The most common implementation is the *lowering (or* IM2COL*-based) approach* which transforms the input tensor into an augmented matrix in order to cast the operator in terms of a cache-friendly general matrix–matrix (GEMM) multiplication [3–5]. This approach has the advantage of exploiting the excellent performance of optimized matrix product implementations, but it requires a very large amount of memory. In contrast, the *direct algorithm*, consisting of 6–7 nested loops around a multiply-and-accumulate instruction, provides arguably the simplest implementation while requiring much less extra memory [2].

In this paper, we initially follow the ideas in [6] to design a cache-friendly, GEMM-like formulation of the direct convolution, making the following new contributions[1] with respect to that work:

- In contrast with [7], our solution preserves the standard NHWC layout for the input tensor and only requires specialized packing for the filter tensor. To achieve this, (1) we decouple the dimension of the microkernel from the cache configuration parameters, which has the positive side effect of improving performance; and (2) we integrate a row-wise oriented microkernel, similar to that introduced in [8].
- In addition, we propose an alternative blocked variant which, during the execution of the convolution operator, packs a small piece of the input tensor into a buffer, in order to accesses with ensure unit stride memory to both the filter and input tensor during the microkernel execution. This approach permits the use of the architecture-specific microkernels for GEMM in the BLIS library.
- Third, we conduct a complete performance analysis of the two new algorithms on an ARMv8 core for two relevant convolutional DNNs (GoogleLeNet and ResNet-50 v1.5), combined with the ImageNet dataset, showing their advantages in comparison with high-performance implementations of the convolution operator based on the IM2COL approach and the Winograd transform. Concretely, our experiments with GoogleNet and ResNet-50 show that new algorithms are 13% to 85% faster than the traditional IM2COL-based approach.

The rest of the paper is structured as follows. Section 2 provides a brief summary of different convolution algorithms and optimization approaches for the direct convolution variant. In Section 3 we review

* Corresponding author.
E-mail address: dolzm@uji.es (M.F. Dolz).

[1] The source code of the direct convolution variants presented in this paper is available at https://github.com/hpca-uji/convDirect.

https://doi.org/10.1016/j.sysarc.2022.102806
Received 13 May 2022; Received in revised form 25 November 2022; Accepted 12 December 2022
Available online 15 December 2022

the basics of the convolution operator and several scalar and blocked algorithms for its computation, including the high-performance implementation proposed in [7]. Next, in Section 4, we present our new two NHWC-preserving algorithms for the direct convolution, emphasizing the differences with respect to previous solutions. Finally, in Section 5 we report the results from a complete evaluation, and in Section 6 we close the paper with a few concluding remarks.

2. Related work

Among the different methods proposed in the literature for the convolution operator, we can list (1) the *direct algorithm*, usually implemented as six nested loops around a multiply-and-add instruction [7]; (2) the *lowering* approach, which applies either IM2COL or IM2ROW to the activation input matrix to then obtain the result from a GEMM [3,4]; (3) the *FFT-based algorithm*, which shifts the computation into the frequency domain in order to reduce the arithmetic requirements [9–11]; and (4) the *Winograd-based convolution*, which leverages the Winograd minimal filtering algorithm to decrease the arithmetic cost of the convolution [9,12].

With regards to the direct algorithm for the convolution, the implementation by Zhang et al. [7] proposes a reorganization of the algorithm loops around a microkernel, mimicking the conventional implementation of high-performance instances of GEMM in libraries such as GotoBLAS, OpenBLAS, and BLIS [13–15]. That work also includes a specialized, cache-friendly data layout for both the input and filter tensors to operate with standard data layouts (e.g., NCHW or NHWC). For that, however, the input/output tensors have to be transformed to/from that format before the first DNN layer and after the last one, respectively. This need for a specialized layout may offer a reason why the direct convolution algorithm is not a widely-adopted. To deal with that, in this work we refine this algorithm to preserve the NHWC input/output tensor format.

In this line, we can also find optimization approaches for the direct convolution algorithm based on exploiting data reuse and the design of near I/O-optimal data layout strategies. For instance, the work by Zhang et al. [16], applies an aggressive design of auto-tuning based on I/O lower bounds which outperform the optimal solutions by TVM. Similarly, the work by Li et al. [17] introduces a block hardware-friendly direct convolution implementation that can completely avoid the off-chip memory transfers of intermediate feature maps on memory-limited devices, e.g., FPGAs (field programmable gate arrays) or MCUs (micro-controller units).

Other trends for deriving optimized direct convolution implementations use meta-programming techniques, such as those proposed by Zlateski et al. in [18], or auto-tuning compilation stacks, such as Apache TVM [19]. For instance, the work by Liao et al. [20] uses TVM to analyse the sparsification opportunities for the acceleration of the direct convolution by pruning certain some CNNs weights. In a similar line, the work by Georganas et al. [21] proposes JIT-based optimization approaches for the direct convolution kernels on x86 architectures.

3. Algorithms for the direct convolution

3.1. Simple implementation of the direct convolution

Let us consider the convolution operator

$$O = \text{CONV}(F, I), \tag{1}$$

where I is a 4D input tensor consisting of N input images of size $H_i \times W_i \times C_i$ each, $H_i \times W_i$ denote the image height × width, and C_i stands for its number of input channels (or features). Furthermore, F is a 4D input tensor consisting of C_o filters of height × width × channels = $H_f \times W_f \times C_i$ each. The operator in (1) then produces a 4D output tensor O, with N outputs of size $H_o \times W_o \times C_o$ each, where $H_o \times W_o$ stand for the output height × width, and C_o for the number of output channels

```
1  void ConvDirect_Naive( I[N][H_i][W_i][C_i],
2                         F[H_f][W_f][C_i][C_o],
3                         O[N][H_o][W_o][C_o] )
4  {
5    for ( h=0; h<N; h++ )
6      for ( i=0; i<C_i; i++ )
7        for ( j=0; j<C_o; j++ )
8          for ( k=0; k<W_o; k++ )
9            for ( l=0; l<H_o; l++ )
10             for ( m=0; m<W_f; m++ )
11               for ( n=0; n<H_f; n++ )
12                 O[h][l][k][j] += I[h][l+n][k+m][i] * F[n][m][i][j];
13 }
```

Listing 1: Naive algorithm for the direct convolution.

(which, as stated previously, is equal to the number of filters). For this purpose, each of the C_o individual filters combines (or 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. By repeatedly applying the filter to the whole input, in a sliding window manner, the convolution operator produces the complete entries of this single output; see [2] and Fig. 1. For simplicity, hereafter we will consider that the filter is applied with unit vertical/horizontal strides; and the output is not padded so that $H_o = H_i - H_f + 1, W_o = W_i - W_f + 1$.

A naive algorithm that computes the convolution is given in Listing 1. Overall, the algorithm traverses the dimensions of the convolution operator in the order $N \to C_i \to C_o \to W_o(W_i) \to H_o(H_i) \to W_f \to H_f$, from the outermost to the innermost loop, which results in a certain pattern of memory accesses depending on the layout of the tensors in memory. Here we will assume that the operands are laid out following a generalization of the row-major matrix format for the multidimensional tensors, which implies that the dimensions are closer in memory from right to left (closest to farthest; see Fig. 2). Furthermore, we will adopt the notation

$$I_{\text{MEM}}[N][H_i][W_i][C_i], \quad O_{\text{MEM}}[N][H_o][W_o][C_o].$$

to specify this layout in memory for the 4D input/output tensors. Note that, this corresponds to the standard NHWC format for 4D tensors in deep learning (DL). Also, the 4D filter tensor is stored following the standard RSCK format; that is,

$$F_{\text{MEM}}[H_f][W_f][C_i][C_o].$$

We close this short review of the direct convolution by highlighting that the 7 loops in the algorithm are independent of each other as well as from the memory layout of the input, output and filter tensors. In consequence, the algorithm loops can be reorganized in any order while still producing the correct result.

3.2. Blocking the direct convolution

In [7], the authors propose a complete reorganization of the naive algorithm for the direct convolution in Listing 1 with the enhancements described in the following paragraphs.

Reordering the loops to saturate computations. The two innermost loops of the algorithms are chosen to traverse the C_o and W_o dimensions of the convolution operator (from innermost to outermost) in order to:

1. Favor that the processor floating-point arithmetic units (FPUs) experience no stalls due to data dependencies between consecutive writes to the same entry of the output tensor; and
2. Accommodate SIMD (single instruction, multiple data) vectorization in the innermost loop by exploiting the number of channel outputs, C_o. (As C_o is a user-specified parameter, it can be selected to be a multiple of the size of the SIMD vector.)

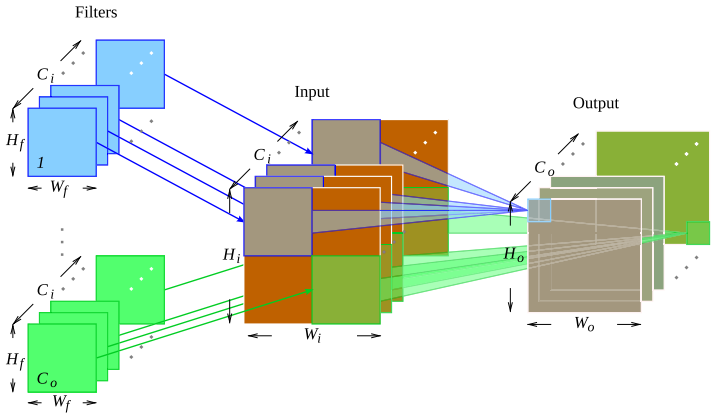

Fig. 1. Sliding window of C_o filters over an input image producing the output of the convolution operator for each output channel.

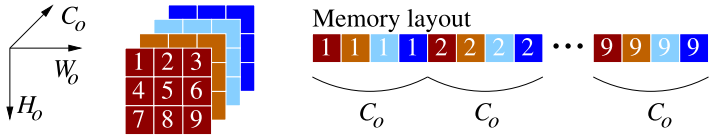

Fig. 2. Memory layout for the 4D output tensor O assuming $N = 1$ and a generalization of the conventional row-major layout for matrices and the NHWC format for the tensor. The same layout applies to the 3D input tensor I.

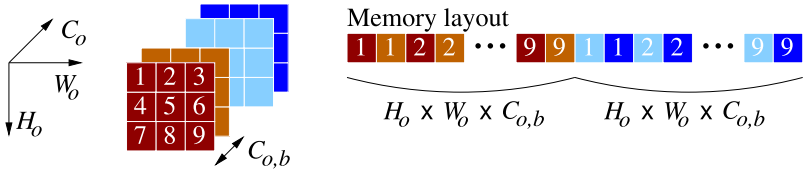

Fig. 3. Convolution-friendly memory layout for the 4D output tensor proposed in [7], assuming $N = 1$.

```
void ConvDirect_Reordered( I[N][H_i][W_i][C_i],
                           F[H_f][W_f][C_i][C_o],
                           O[N][H_o][W_o][C_o] )
{
  for ( h=0; h < N; h++ )
    for ( l=0; l < H_o; l++ )
      for ( n=0; n < H_f; n++ )
        for ( m=0; m < W_f; m++ )
          for ( i=0; i < C_i; i++ )
            for ( k=0; k < W_o; k++ )
              for ( j=0; j < C_o; j++ )
                O[h][l][k][j] += I[h][l+n][k+m][i] * F[n][m][i][j];
}
```

Listing 2: Algorithm for the direct convolution with re-ordered loops.

Ordering the loops to optimize data reuse. After having reordered the two innermost loops, the next three loops in the sequence from innermost to outermost are set to traverse the C_i, W_f, H_f dimensions of the operator so as to ensure that the data from the input and output tensors are accessed in the same order. For convolutional DNNs, this is beneficial because the output of one convolution layer is often the input to the next one. The initial algorithm with the reordered loops in [7] is reproduced in Listing 2.

Blocking for the memory hierarchy. In this stage:

1. Due to the limited number of vector registers in any processor architecture, tiling [22] with block sizes $C_{o,b}, W_{o,b}$ is applied to the two innermost loops of the reordered algorithm. This is performed to avoid register spilling that can result in degraded performance.
2. Furthermore, the loop traversing the $W_o/W_{o,b}$ dimension is placed between the H_o and H_f loops in order to mimic the ordering of the loops traversing the filter weights.
3. Also, the loop traversing the $C_o/C_{o,b}$ dimension is set as the outermost loop to facilitate parallelization.
4. In addition, the C_i dimension of the operator is partitioned with block size $C_{i,b}$ so as to split the input dataset into smaller blocks that fit into the memory hierarchy.

The resulting blocked algorithm in [7] is illustrated in Listing 3.

Convolution-friendly data layout. To ensure that the output of the convolution operator is accessed with unit stride:

1. The output tensor is rearranged in memory into sequential blocks of $N \times H_o \times W_o \times C_{o,b}$ elements, where the entries of each block are first arranged in the channel dimension, and then following the row-major order into $H_o \times W_o$ blocks of length $C_{o,b}$.
2. For compatibility between the outputs and inputs of consecutive layers in convolutional DNNs, the same layout applies to the input tensor; see Fig. 3. In our notation, this corresponds to:

$$I_{\text{MEM}}[N][C_i/C_{i,b}][H_i][W_i][C_{i,b}],$$
$$O_{\text{MEM}}[N][C_o/C_{o,b}][H_o][W_o][C_{o,b}].$$

3. Finally, the filter tensor is rearranged in memory with its dimensions organized as [7]:

$$F_{\text{MEM}}[C_o/C_{o,b}][C_i/C_{i,b}][H_f][W_f][C_{i,b}][C_{o,b}].$$

```
void ConvDirect_Blocked( I[N][H_i][W_i][C_i],
                         F[H_f][W_f][C_i][C_o],
                         O[N][H_o][W_o][C_o] )
{
  for ( h = 0; h < N; h++ )
   for ( j' = 0; j' < C_o/C_o,b; j'++ )
    for ( i' = 0; i' < C_i/C_i,b; i'++ )
     for ( l = 0; l < H_o; l++ )
      for ( k' = 0; k' < W_o/W_o,b; k'++ )
       for ( n = 0; n < H_f; n++ )
        for ( m = 0; m < W_f; m++ )
         // Micro-kernel for C+=A*B
         for ( ii = 0; ii < C_i,b; ii++ )
          for ( kk = 0; kk < W_o,b; kk++ )
           for ( jj = 0; jj < C_o,b; jj++ )
            O[h][l][k' · W_o,b + kk][j' · C_o,b + jj]
                 += I[h][l + n][k' · W_o,b + kk + m][i' · C_i,b + ii]
                  * F[n][m][j' · C_o,b + jj][i' · C_i,b + ii]
}
```

Listing 3: Blocked algorithm for the direct convolution.

```
void ConvDirect_New( I[N][H_i][W_i][C_i],
                     F[H_f][W_f][C_i][C_o],
                     O[N][H_o][W_o][C_o] )
{
  // Loops for h, j', i', l, k', n
  // remain the same as in the blocked
  // algorithm in Listing 3 and therefore
  // they are omitted for brevity
  for ( m = 0; m < W_f; m++ )
   for ( kk = 0; kk < W_o,b; kk += M_r )
    for ( jj = 0; jj < C_o,b; jj += N_r )
     // Micro-kernel
     for ( ii = 0; ii < C_i,b; ii++ )
      for ( j_r = kk; j_r < kk + M_r; j_r++ )
       for ( i_r = C_o,b; i_r < C_o,b + N_r; i_r++ )
        O[h][l][k' · W_o,b + j_r][j' · C_o,b + i_r]
             += I[h][l + n][k' · W_o,b + j_r + m][i' · C_i,b + ii]
              * F[n][m][j' · C_o,b + i_r][i' · C_i,b + ii]
}
```

Listing 4: Blocked algorithm for the direct convolution with decoupled micro-kernel loops.

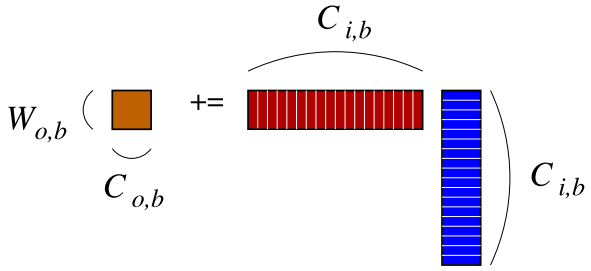

Fig. 4. Microkernel for $C += A \cdot B$. The microtile C (in brown) is initially retrieved into the processor registers and updated at each the $C_{i,b}$ iterations of a loop via an outer product between a column of the micropanel A (in red) and a column of the micropanel B (in blue). Upon completion, C is copied back into the memory. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

GEMM-*based microkernel.* The three innermost loops in the blocked algorithm for the convolution, traversing the $C_{i,b}, W_{o,b}, C_{o,b}$ dimensions of the operator (see Listing 3), perform a small GEMM, of the form $C += A \cdot B$, where C is a $W_{o,b} \times C_{o,b}$ microtile of O; A is a $W_{o,b} \times C_{i,b}$ micropanel of I; and B is a $C_{i,b} \times C_{o,b}$ micropanel of F. Following the usual implementation of this type of microkernels in high-performance instances of the BLAS (basic linear algebra subprograms [23]), this is done by first loading the contents of the microtile C into the processor (vector) registers, to then update these elements via a sequence of $C_{i,b}$ outer products, each involving a single column/row of the micropanels A/B consisting of $W_{o,b}/C_{o,b}$ elements; see [15,24] and Fig. 4. For high performance, vector fused-multiply-add (FMAs) instructions are used to perform these updates using the processor SIMD FPUs.

The size of the microkernel integrated in the direct convolution is determined based on the latency and throughput of these type of instructions, as well as on the available number of architecture vector registers. Specifically, given a fixed number of SIMD registers available on the target processor, the design of the microkernel seeks to maximize the number of output elements computed while staying within the fixed number of available registers.

The layout chosen for the output (O) and filter tensors (F) in [7] ensures that the entries of the microtile C and the rows of the micropanel B can be accessed from the microkernel with unit stride. In contrast, the layout of the input (I) implies that the micropanel A is stored with consecutive columns but with a row stride (or row leading dimension) `ldA`$= C_{i,b}$. For the microkernel, this results in non-unit stride accesses to the columns of the micropanel A. However, $C_{i,b}$ can be chosen to be small enough to ensure that the consecutive elements reside in the same cache line.

4. New blocked NHWC-preserving algorithms for the direct convolution

In this paper, we build upon the work in [7] by maintaining part of the blocking strategy proposed there, but making the major changes described in the following paragraphs.

Decouple the microkernel dimension from the cache blocking parameters. The blocked algorithm in [7] ties the blocking parameters $C_{i,b}, W_{o,b}, C_{o,b}$ to the dimensions of the small GEMM performed inside the microkernel. In our work, we avoid this by introducing two additional loops that traverse the $W_{o,b}, C_{i,b}$ dimensions of the operator in steps of size M_r, N_r respectively, as shown in Listing 4. As a result, the microkernel now performs a small GEMM of the form $C += A \cdot B$, where C is a microtile of size $M_r \times N_r$, and A, B are micropanels of size $M_r \times C_{i,b}$ and $C_{i,b} \times N_r$, respectively.

Preserving the input/output NHWC layout. In contrast with [7], we preserve the original NHWC layout for the input and output tensors. Accesses with non-unit stride to C from the microkernel can be expected to be well amortized due to the $C_{i,b}$ ratio between the number of memory accesses to load/store the microtile C ($2M_rN_r$ memory accesses) and the number of floating-point arithmetic operations ($2M_rN_rC_{i,b}$ flops) to update it. The costs of accessing B and A from the microkernel are respectively tackled by arranging the filter tensor following a particular layout in memory, and preparing the data for the microkernel, as detailed next.

Ensuring unit-stride accesses to B: data layout for filter tensor. We aim at maintaining a micropanel B, of dimension $C_{i,b} \times N_r$, in the L1 cache during the execution of the microkernel by arranging the entries of the filter tensor by blocks (tiling), as:

$$F_{\text{MEM}}[H_f][W_f][C_o/C_{o,b}][C_i][C_{o,b}],$$

combined with an appropriate selection of the cache configuration parameters $C_{i,b}, N_r$, which match the size of the L1 cache and its associativity degree [24]. This has the additional advantage of ensuring that the micropanel of B is properly arranged in memory so as to accommodate unit-stride accesses from the microkernel. Also, the cost

```
#define VFMA(Creg, Breg, Creg, offset)\
        Creg = vfmaq_laneq_f32(Creg, Breg, Areg, offset)
#define VLD    vld1q_f32
#define VST    vst1q_f32

#define ACCESS_C(VOP) {\
  VOP(&C[0],     C00); VOP(&C[4],       C01);\
       VOP(&C[8],       C02);\
  // ... Omitted for brevity\
  VOP(&C[ldC*6], C60); VOP(&C[4+ldC*6], C61);\
      VOP(&C[8+ldC*6], C62);}

#define LOAD_B(j) {\
  B0 = VLD(&Bptr[j]); B1 = VLD(&Bptr[4+j]);\
           B2 = VLD(&Bptr[8+j]);}

#define UPDATE_C(j) {\
  VFMA(C00,B0,A0,j); VFMA(C01,B1,A0,j); VFMA(C02,B2,A0,j);\
  // ... Omitted for brevity \
  VFMA(C60,B0,A6,j); VFMA(C61,B1,A6,j); VFMA(C62,B2,A6,j);}

// Load micro-tile of C
ACCESS_C(VLD)

// Iterate inside loop from kr = 0 to Cib-3
baseB = 0;
for ( kr = 0; kr < Cib-3; kr += 4 ) {
 // Load 7 x 4 block of A
 A0 = VLD(&A[kr]);
 // ... Omitted for brevity
 A6 = VLD(&A[kr+ldA*6]);
 // Load 1x12 row of B; Update micro-tile w.r.t column of A
 LOAD_B(baseB);   UPDATE_C(0);LOAD_B(baseB+12);UPDATE_C(1);
 LOAD_B(baseB+24);UPDATE_C(2);LOAD_B(baseB+36);UPDATE_C(3);
 baseB += 4*Nr;
}

// Last iterations, from kr+1 to Cib-1, omitted for brevity
// Store micro-tile of C
ACCESS_C(VST)
```

Listing 5: Structure of the 7×12 microkernel for algorithm A, implemented using ARM NEON intrinsics.

of this reorganization is low and only needs to be done once, as the filters of a DNN model do not vary during the inference process.

We next describe two distinct strategies to deal with the non-unit accesses to the elements of A from the microkernel, leading to two distinct algorithms, described in the next two subsections.

4.1. New Algorithm A

Amortizing the cost of non-unit stride accesses to A with a specialized microkernel. To illustrate this technique in detail, via a concrete example, let us consider the IEEE single-precision floating-point (FP32) format as the baseline datatype for the tensor data and arithmetic, and consider a target ARMv8 architecture with 32 (SIMD) vector registers of width 128 bits (i.e., with capacity for 4 FP32 numbers). We can follow [8] to implement a microkernel that operates with a microtile of dimension $M_r \times N_r = 7 \times 12$, while unrolling the loop that traverses the $C_{i,b}$ dimension with a factor 4, with the following implications:

- 21 (7×3) vector registers are employed to store the entries of the microtile C. As each vector register can store 4 FP32 numbers, this results in the required $7 \times (3 \cdot 4) = 7 \times 12$ microtile.
- 7 vector registers are dedicated to storing 4 entries each of 7 consecutive rows of A.
- 3 vector registers are employed to store $(3 \cdot 4) = 12$ entries of B.
- Each iteration of the microkernel uploads 7 rows of A, to then repeat four times the following: load a row of B (12 elements) and then update the microtile (7×12) C with respect to one of the block columns uploaded from A and the row of B.
- Vector FMAs are used for the updates.

The general structure of the microkernel is illustrated in Listing 5, showing the use of ARM NEON intrinsics for the vector loads, stores and FMAs.

Considering the data layout for the data tensors, we remark the following details:

- As corresponds to the NHWC format, the entries of the microtile C are stored in row-major order with row stride `ldC`$= C_o$. The complete microtile is loaded before the microkernel loop, using 21 vector instructions, and stored after it, also with 21 vector instructions. The costs of these memory accesses are amortized over the $C_{i,b}$ iterations of the loop.
- For compatibility, the entries of the micropanel A are also stored in row-major order, in this case with row stride `LDA`$= C_i$. By unrolling the loop with a factor 4, at each iteration of the loop in the microkernel and for each row of A we upload 4 consecutive entries of the micropanel using a vector instruction. This

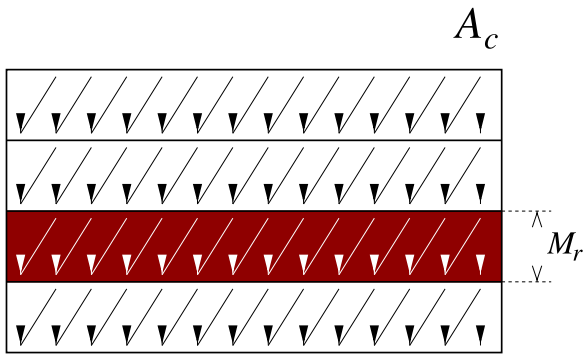

Fig. 5. Memory layout for the $W_{o,b} \times C_{i,b}$ buffer A_c.

compensates the cost of accessing non-consecutive rows of A in memory.
- The entries of the micropanel B are stored in row-major order, with row stride $N_r = 12$. Each iteration of the loop in the microkernel retrieves $(4 \cdot 12) = 48$ consecutive entries of the panel, yielding an access with unit stride.

Other microkernel dimensions. This technique is indeed not exclusive for the $M_r \times N_r = 7 \times 12$ microkernel. For example, the same principles can be applied to an $M_r \times N_r = 6 \times 16$ microkernel where:

- 24 (6×4) vector registers are employed to store the microtile of C.
- 6 vector registers are dedicated to 4 entries each of 6 consecutive rows of A.
- 2 vector registers are employed to store $(2 \cdot 4) = 8$ entries of B, but these vector registers are reutilized four times during the unrolled loop iteration.

This yields a total utilization of 32 vector registers in the variant with a 6×16 microkernel versus the 31 vector registers employed in the 7×12 microkernel.

In general, the dimensions of the micro-kernel are constrained by the number of vector registers in the processor architecture. Here we follow the principles in [24] to select the dimensions that maximize the usage of vector registers without incurring in register spilling during the execution of the micro-kernel. This led us to choose several configurations among which, we will only report results for those that delivered the best general performance for each algorithm.

Amortizing the cost of loading/storing the microtile of C. We next note that the microtile of C updated in the new Algorithm A is independent of the loops traversing the H_f, W_f filter dimensions (indexed by n, m in our algorithms). Therefore, it is possible to move these two loops inside the microkernel, so as to amortize the costs of these accesses to the C operand, which is expected to reside in the main memory and, therefore, can be expensive to access.

4.2. New Algorithm B

Ensuring unit-stride accesses to A with via packing. The previous algorithm requires a specialized microkernel, different from that available in the BLIS library for matrix multiplication. Developing this microkernel for each particular processor architecture can be a complex and costly process, as this is usually done using assembly instructions to carefully prefetch the data into the processor vector registers in order to completely overlap communication with computation [15,25,26].

The alternative strategy adopted in the algorithm proposed next follows the BLIS strategy to pack a $W_{o,b} \times C_{i,b}$ block of $D = A$ into a temporary buffer A_c, in blocks of M_r rows as illustrated in Fig. 5. This packing is done inside the loop that traverses the W_o dimension of the problem in Listing 4. For illustrative purposes, Listing 6 shows a BLIS-like implementation of an 8×12 microkernel for algorithm B that assumes that A is packed into a contiguous buffer A_c. There:

- 24 (8×3) vector registers are employed to store the entries of the 8×12 microtile C.
- 2 vector registers are dedicated to store 8 consecutive entries of a column of A_c.
- 3 vector registers are employed to store 12 entries of a row of B.
- Each iteration of the microkernel uploads a column of the buffer A_c and 12 entries in a row of B to then update the 8×12 microtile of C with respect to them.
- Vector FMAs are used for the updates.

Compared with all the previous algorithms for the direct convolution, including our new algorithm A, on the negative side:

1. This variant introduces a memory overhead due to the additional workspace for the A_c buffer.
2. There is some cost due to the copying of the data into A_c.

On the positive side:

1. This new algorithm B can leverage the natural BLIS microkernel to perform the small GEMM.
2. The cost of data copying is likely to be well amortized over enough computations in the inner loops.
3. The rearrangement ensures unit-stride accesses to the micropanel of A from the microkernel; see the algorithm in Listing 6.

Algorithm B presents an additional advantage for convolution operators with horizontal and/or vertical strides larger than 1. Concretely, this type of access to the input tensor can be accommodated when packing the entries of the $W_{o,b} \times C_{i,b}$ of A into the buffer A_c so that the high-performance implementation of the microkernel does not need to be modified.

Amortizing the cost of loading/storing the microtile of C. A similar idea to that described at the end of the previous subsection can be applied to Algorithm B. However, that would imply that the use of the BLIS microkernel is no longer an option and, therefore, we do not consider it further for this algorithm.

4.3. Vector intrinsics versus assembly

The implementations of the 7×12 microkernel for Algorithm A and the 8×12 microkernel for Algorithm B, respectively shown in Listing 5 and 6, employed ARM NEON intrinsics. This leaves in the hands of the compiler the decision on (1) how to translate each intrinsic into a specific collection of assembly instructions; and (2) which architecture vector registers to use in order to store each variable, which often results in large variability in the attained performance. This is because the characteristics of individual instructions (e.g., latency and throughput) are significant factors in the design of the microkernel. As such, we chose to implement the microkernel using inline assembly to specifically use certain types of instructions. This is illustrated by redefining the `VFMA` macro utilized in the two microkernels as in Listing 7.

For the second issue, we noticed that the compiler tends to produce code that resulted in a significant amount of register spilling, resulting in superfluous load and store instructions being generated. We overcame this issue by implementing the entire microkernel in assembly. We have done so for some of our simpler cases, including the most used microkernels for Algorithms A and B. Here it is worth mentioning that the development of microkernels is a systematic process, which can be automated to a certain extent with good performance results [27]. We leave the application of this research line to the convolution operator as part of future work.

```
#define VFMA(Creg, Breg, Creg, offset)\
        Creg = vfmaq_laneq_f32(Creg, Breg, Areg, offset)
#define VLD    vld1q_f32
#define VST    vst1q_f32

#define ACCESS_C(VOP) {\
  VOP(&C[0],     C00); VOP(&C[4],       C01);\
      VOP(&C[8],       C02);\
  VOP(&C[ldC],   C10); VOP(&C[4+ldC],   C11);\
      VOP(&C[8+ldC],   C12);\
  // ... Omitted for brevity\
  VOP(&C[ldC*7], C70); VOP(&C[4+ldC*7], C71);\
      VOP(&C[8+ldC*7], C72);}

// Load micro-tile of C
ACCESS_C(VLD)

// Iterate inside loop from kr = 0 to Cib-1
baseA = 0; baseB = 0;
for ( kr = 0; kr < Cib; kr++ ) {
 // Load column of A
 A0 = VLD(&A[baseA]);
 A1 = VLD(&A[baseA+4]);
 // Load row of B
 B0 = VLD(&B[baseB]);
 B1 = VLD(&B[baseB+4);
 B2 = VLD(&B[baseB+8]);
 // Update micro-tile w.r.t column of A and row of B
 VFMA(C00,B0,A0,0); VFMA(C01,B1,A0,0); VFMA(C02,B2,A0,0);\
 VFMA(C10,B0,A0,1); VFMA(C11,B1,A0,1); VFMA(C12,B2,A0,1);\
 // ... Omitted for brevity\
 VFMA(C70,B0,A1,3); VFMA(C71,B1,A1,3); VFMA(C72,B2,A1,3);}
 baseA += Mr; baseB += Nr;
}

// Store micro-tile of C
ACCESS_C(VST)
```

Listing 6: Structure of the 8×12 microkernel for algorithm B, implemented using ARM NEON intrinsics.

```
#define VFMA(Creg, Breg, Areg, offset)\
__asm__ volatile\
(\
  "fmla %[c_reg].4s,"\
       "%[b_reg].4s,"\
       "%[a_reg].s["#offset"] \n\t"\
: [c_reg] "+w" (Creg)\
  [b_reg] "w"  (Breg),\
: [a_reg] "w"  (Areg)\
);\
```

Listing 7: Use of assembly inlining.

5. Experimental results

5.1. Hardware setup

All the experiments in this section were performed using IEEE single-precision (FP32) on a single core of an NVIDIA Carmel (ARMv8.2) processor. This architecture is equipped with a private 4-way associative (data) L1 cache of 64 KiB per core, a 16-way associative L2 cache of 2 MiB shared between each pair of cores, a 16-way associative L2 cache of 4 MiB shared by all 8 cores, and 4 GiB of RAM. To avoid the performance distortions caused by the utilization of distinct power modes (and associated processor core frequencies), the operating core frequency was set to 2.3 GHz. The theoretical peak performance for a single core of this architecture, operating at such frequency, is 36.8 (FP32) GFLOPS (billions of floating-point operations, or flops, per second). Moreover, the single worker thread was pinned to a specific core of the NVIDIA Carmel processor.

There are several approaches to parallelize the convolution algorithms targeted in this work by exploiting multi-threading on a multicore processor. In [28], we evaluate different parallel approaches and conclude that the best strategy is to run one instance of the model per core in the system, and then distribute the batch among the different instances. From that point of view, we believe that an evaluation using a single thread is illustrative of the behavior that could be attained with this parallelization strategy.

5.2. Data setup

For the experiments, we selected the convolution layers present in two popular DNNs: GoogleLeNet [29] and ResNet-50 (v1.5) [2,30]. From these, we discarded those layers with horizontal/vertical strides greater than 1, because they are more difficult to tackle efficiently by the blocked algorithm in [7] and the new algorithm A described in Section 4. Overall, the convolution layers included in the following experimental study represent around 70% of the total execution time of the CNN on the target architecture (though we note that our codes do not exploit other orthogonal optimization techniques, such as fusion,

which could increase the contribution of the convolution operators to the total cost of the inference).

The dataset that was selected in both cases was ImageNet [31] and the batch size was set to $N = 1$ in order to reflect the single stream scenario of the ML Commons benchmark for inference on the edge.[2]

5.3. Algorithms

In the comparison, we include five options for the calculation of the convolution operators appearing in the corresponding layers of the two selected DNNs:

- LOWERING: The lowering approach with the IM2COL transform implemented as an optimized C routine; and the subsequent GEMM performed by invoking BLIS (v0.8.1), with the configuration of this library adjusted to the NVIDIA Carmel processor. (We also ran the same option linked with OpenBLAS version v0.3.19 and ARMPL version v.21.1, in general obtaining lower performance. Therefore, we do not consider their GEMM instances for the LOWERING alternative in the following discussion.)
- BLOCKED: The blocked algorithm for the direct convolution described in [7]. In this case, we developed a microkernel with O resident in registers of dimension $W_{o,b} \times C_{o,b} = 7 \times 12$, following the ideas in [8], and optimized it using ARM NEON intrinsics (see Listing 5). We remind that, because of the connection between consecutive convolution layers, this algorithm enforces the constraint that $C_{i,b} = C_{o,b}$.
- NEW-A: The blocked NHWC-preserving Algorithm A for the direct convolution introduced in Section 4 of this paper, with $M_r \times N_r = 7 \times 12$ and 6×16 microkernels optimized using ARM NEON intrinsics (see, e.g., Listing 5). For this particular case, we also developed our own assembly versions of these two microkernels. For this algorithm, we select a value for $C_{i,b}$ which favors that a $C_{i,b} \times N_r$ micropanel of B (corresponding to a block of the filter tensor) remains in the L1 cache during the execution of the microkernel [24].
- NEW-B: The blocked NHWC-preserving Algorithm B for the direct convolution in Section 4, with the cache configuration parameters selected via the analytical model in [24]. In this case, we either (1) employ our own manually encoded $M_r \times N_r = 8 \times 12$ and 4×20 microkernels using ARM NEON intrinsics; (2) utilize our own equivalent microkernels fully developed in assembly following the approach in Listing 6; or (3) rely on this specific component in BLIS for the ARM processor.
- WINOGRAD: The implementation of this algorithm in the NNPACK library,[3] with a 3×3 filter.

In most of the following plots, the results are reported in terms of GFLOPS, using a cost of $2 C_i C_o H_o W_o H_f W_f$ flops for all the previous algorithmic options. The GFLOPS rate is inversely proportional to the execution time of the algorithm and presents the advantage of being bounded by the peak performance of the target platform (36.8 GFLOPS). The utilization of this count also for the Winograd-based algorithm (even though it performs a considerably smaller number of flops,) allows a direct comparison with the other approaches. The execution of each implementation was repeated for at least 2 minutes, and the results were then averaged.

We do not include results for the (Apache) AutoTVM tool TVM because optimizing the convolution operator using it requires a considerable execution time. Concretely, AutoTVM explores about 1,000 options per convolution layer, resulting in between 40 min and 1 h to optimize each layer. In contrast, our direct convolution algorithms are straight-forward to tune for a particular architecture, for almost any convolutional layer, independently of its specific parameters. Given these reasons, we prefer to omit the comparison against TVM as we would like to preserve the focus of the paper on those optimizations which can be obtained by mimicking the techniques usually applied in linear algebra libraries for the matrix multiplication.

5.4. Performance evaluation

Microkernels for the new variants. The performance of NEW-A and NEW-B strongly depends on the implementation of the underlying microkernel. For this particular work, due to the dimensions of the convolution operators appearing in the two target models, we "manually" developed microkernels that employ ARM NEON intrinsics (or, alternatively, assembly instructions) for the two variants, with $M_r \times N_r = 7 \times 12$ (see Listing 5) and 6×16 for NEW-A; and $M_r \times N_r = 8 \times 12$ (see Listing 6) and 4×20 for NEW-B. In addition, for NEW-B, we consider the native microkernel for ARM from BLIS (which also sets $M_r \times N_r = 8 \times 12$). As an initial point for our evaluation, we therefore assess the impact of the selected microkernel on the performance of the variants. The results in Figs. 6 and 7 for NEW-A show that the 6×16 microkernel is consistently the best option for GoogleLeNet-ImageNet and dominates the performance in 24 out of 46 layers for ResNet-50-ImageNet. (The layers of the GoogleLeNet-ImageNet and ResNet-50-ImageNet models were respectively split into 3 and 2 plots in these figures to improve visibility.) For NEW-B, Figs. 8 and 9 report the clear performance gain when using the 8×12 microkernel for GoogleLeNet-ImageNet model and, in general, superior performance for the same microkernel for ResNet-50-ImageNet (in 31 out of 46 layers). In addition, for variant NEW-B, the manual microkernels offer superior performance to that observed when using the native implementation in BLIS. This may be surprising but in an independent experiment we could deduce that the reason for this behavior is that the BLIS microkernel is highly optimized when the actual dimensions of the microtile that is to be processed, say $m_r \times n_r$, match the dimensions of the native microkernel. However, the BLIS implementation suffers a serious drop in performance for the "border" cases, when $m_r < M_r = 8$, $n_r < N_r = 12$, or both.

To increase the readability of the following plots, hereafter we will refer with the labels NEW-A or NEW-B to the implementations that respectively integrate the optimal microkernel into Algorithm A or B, which depends on the particular model and layer.

Winograd. While we recognize that the Winograd-based convolution may offer competitive performance, depending on the layer properties, this alternative approach reduces the number of arithmetic operations at the cost of sacrificing numerical accuracy and flexibility. Therefore, a comparison based only on raw performance is incomplete. Nonetheless, we evaluated the Winograd-based algorithm in NNPACK for ResNet-50 model. The first aspect to note is that the package could be used with very few convolution layers of the model: 17 out of 46 (around 37% only). The second issue is that the performance in all these cases varied between 7.5 and 9.5 GFLOPS, therefore being much lower than that observed for the direct convolution implementations applied to the ResNet-50 model with ImageNet presented earlier. Therefore, we exclude the Winograd-based algorithm from the global evaluation in the following experiments.

Global comparison. Figs. 10 and 11 report the performance rates for the distinct implementations of the convolution operator and the two DNN models considered in this work. These results show fair benefits for the NEW-A and NEW-B implementations, which outperform the other options for most layers of both models.

In particular, for the GoogleLeNet-ImageNet pair, the new variants NEW-B and NEW-A outperform the (best) lowering approach by an average speed-up of 1.22 and 1.09 respectively. For NEW-B, the highest speed-up is 1.88 (layer #84) and the lowest one is 0.49 (layer #1). In comparison, for NEW-A the highest speed-up is 1.85 (layer #14) and the lowest one is 0.5 (layer #1).

[2] See https://mlcommons.org/.

[3] https://github.com/Maratyszcza/NNPACK

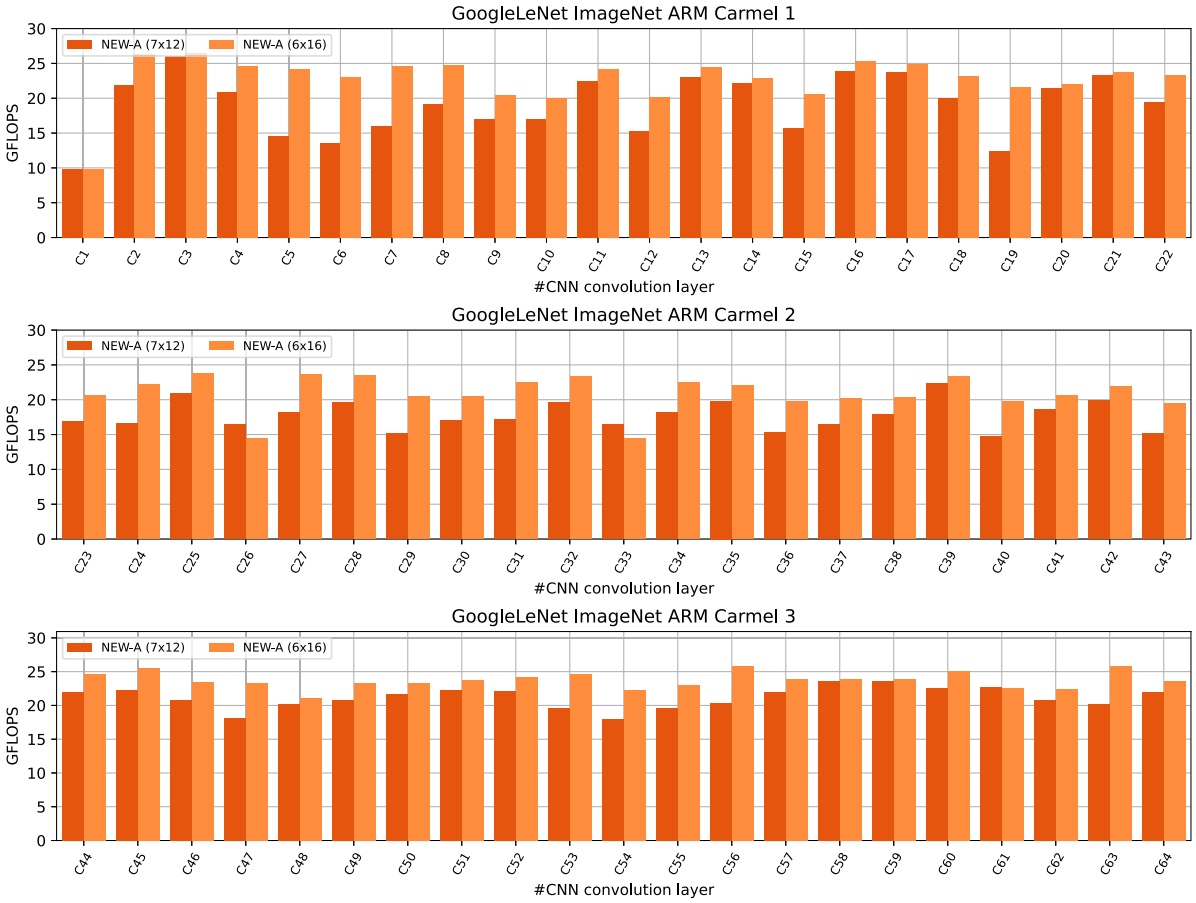

Fig. 6. Performance evaluation of the NEW-A variant for GoogleLeNet.

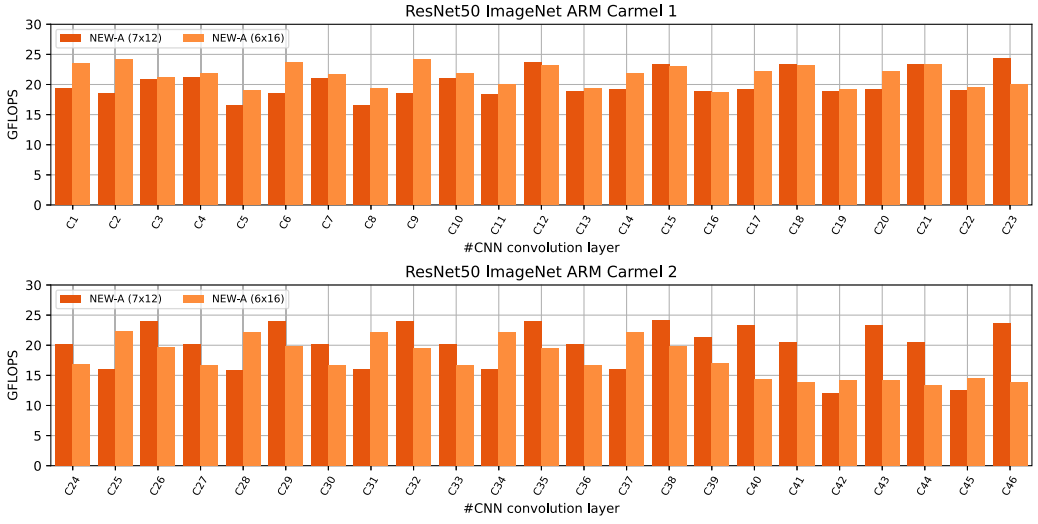

Fig. 7. Performance evaluation of the NEW-A variant for ResNet-50.

For RestNet-50, the average speed-up of NEW-A/NEW-B with respect to the (best) lowering approach is 1.08 and 1.23 respectively, with the highest speed-up being 1.87 (layer #167) and the lowest one 0.90 (layer #92) for NEW-B; and 1.62 (layer #160 and #170) and 0.79 (layer #80) for NEW-A, respectively.

While the speed ups attained for the individual layers for GoogleLeNet-ImageNet or ResNet-50-ImageNet can be regarded as a useful piece of information, to put them into perspective we need to consider their impact on the cost of each layer. In order to do this, we integrated the distinct convolution algorithms into our PyDTNN framework for deep learning [32,33], and performed a complete evaluation of the inference process. Fig. 12 displays the absolute (aggregated) time of the two options: LOWERING and the NEW-B algorithm. (The NEW-A algorithm could not be considered in this comparison as it neither

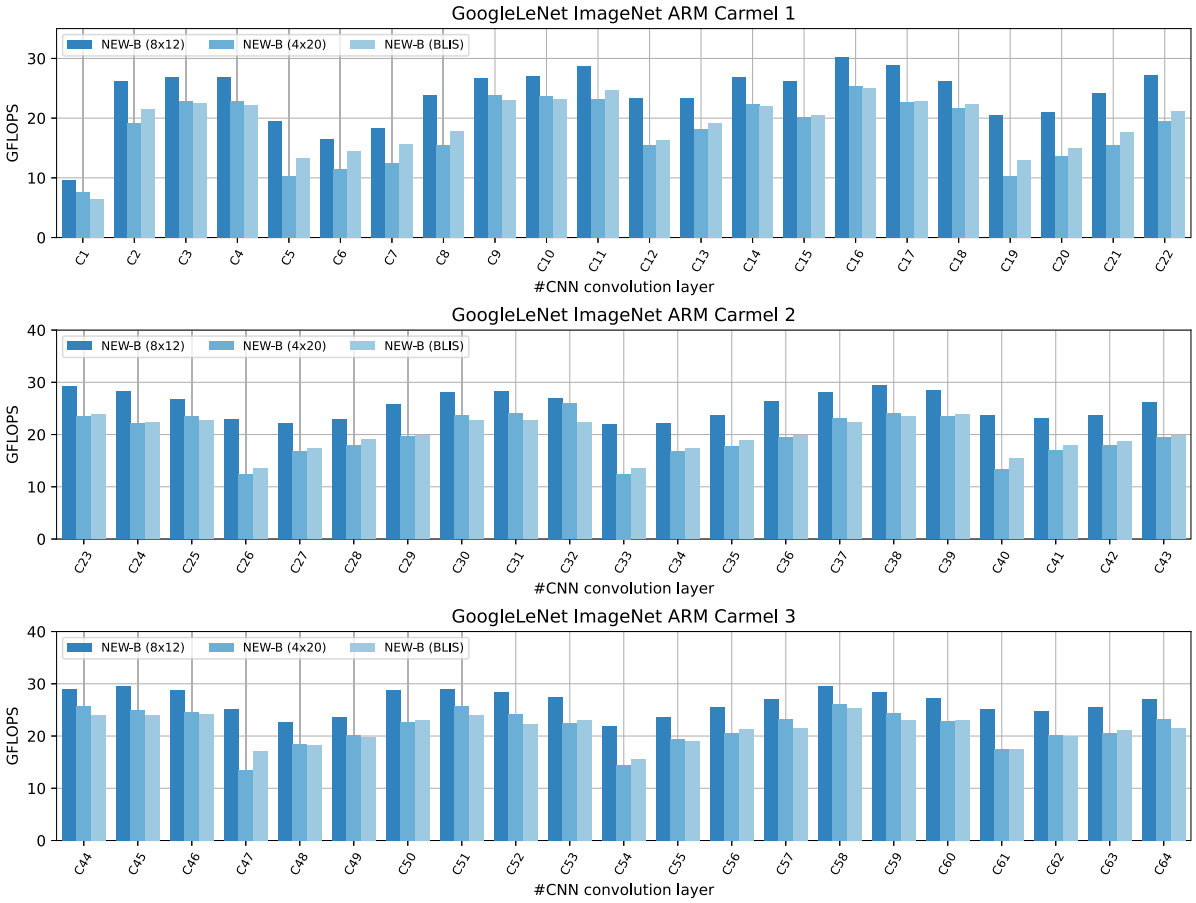

Fig. 8. Performance evaluation of the NEW-B variant for GoogleLeNet.

supports padding nor stride, which are required by the evaluated CNNs.[4]) These results show that NEW-B offer an overall speed-up factor of over 1.25× over LOWERING for the GoogleLenet-ImageNet case and 1.22× for the ResNet-50-ImageNet model. For reference, the line labeled as "Best" in the figure corresponds to an option that chooses the fastest algorithm between LOWERING and NEW-B on a per-layer basis.

We have also performed a similar analysis from the point of view of energy consumption using power the hardware counters that are available in the NVIDIA Carmel processor. The results of this study show a trend for the energy savings which mimic closely that of the reduction in execution time reported in Figs. 12.

5.5. Memory consumption

The convolution methods also differ in the workspace they require. The main advantage of the BLOCKED algorithm and the NEW-A variant is that they both incur a zero-memory overhead. In contrast, the NEW-B counterpart needs a small yet non-negligible workspace, of dimension

$$\min(W_{o,b}, \max_l(W_o^l)) \times \min(C_{i,b}, \max_l(C_i^l)),$$

where W_o^l and C_i^l respectively denote the output width and number of input channels of the convolutional layer l. Here, the blocking parameters $W_{o,b}$, $C_{i,b}$ are experimentally determined to maximize performance.

The LOWERING algorithm in principle requires a workspace of size $\max_l(W_f^l) \cdot \max_l(H_f^l)$ times than that of the largest input tensor I at any layer. (In practice, this can be reduced by applying the IM2COL transform by blocks and interleaving it with the matrix multiplication. On the negative side, this will have an impact on performance, as the multiplications will now involve smaller matrix operands.)

Finally, the memory requirements of the Winograd algorithm are very implementation-dependent. Given that we use the realization of this operator in a external library (NNPACK), we prefer not to theorize over its memory usage.

Table 1 illustrates the memory consumption of the LOWERING and NEW-B algorithms. For perspective, we also include there the memory required to store the model parameters for the two use cases.

Table 1
Memory requirements (in Mbytes) to store the DNN models and the workspaces for the LOWERING and NEW-B methods.

DNN	Model	LOWERING	NEW-B
ResNet-50 v1.5	32.06	6.89	0.05
GoogleLeNet	1049.29	220.50	0.23

6. Concluding remarks and future work

The convolution is an important operation for inference (and training) of DNNs, especially relevant in computer vision and signal processing tasks. While there exist several approaches to implementing this

[4] Both padding and stride can be supported in algorithm New-A. However, this will have a strong negative impact on performance. For example, with a stride different from one, the loads of elements from A inside the loop of the micro-kernel cannot be vectorized. This may result in the micro-kernel being memory-bound instead of compute-bound, resulting in very low performance. A similar effect can arise from padding. For that reason, we decided not to offer support for this in the New-A variant.

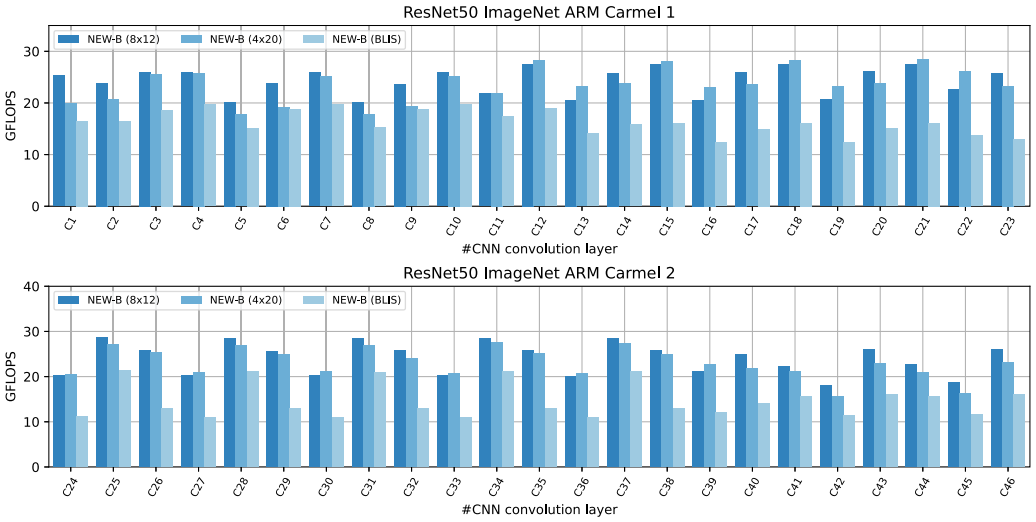

Fig. 9. Performance evaluation of the NEW-B variant for ResNet-50.

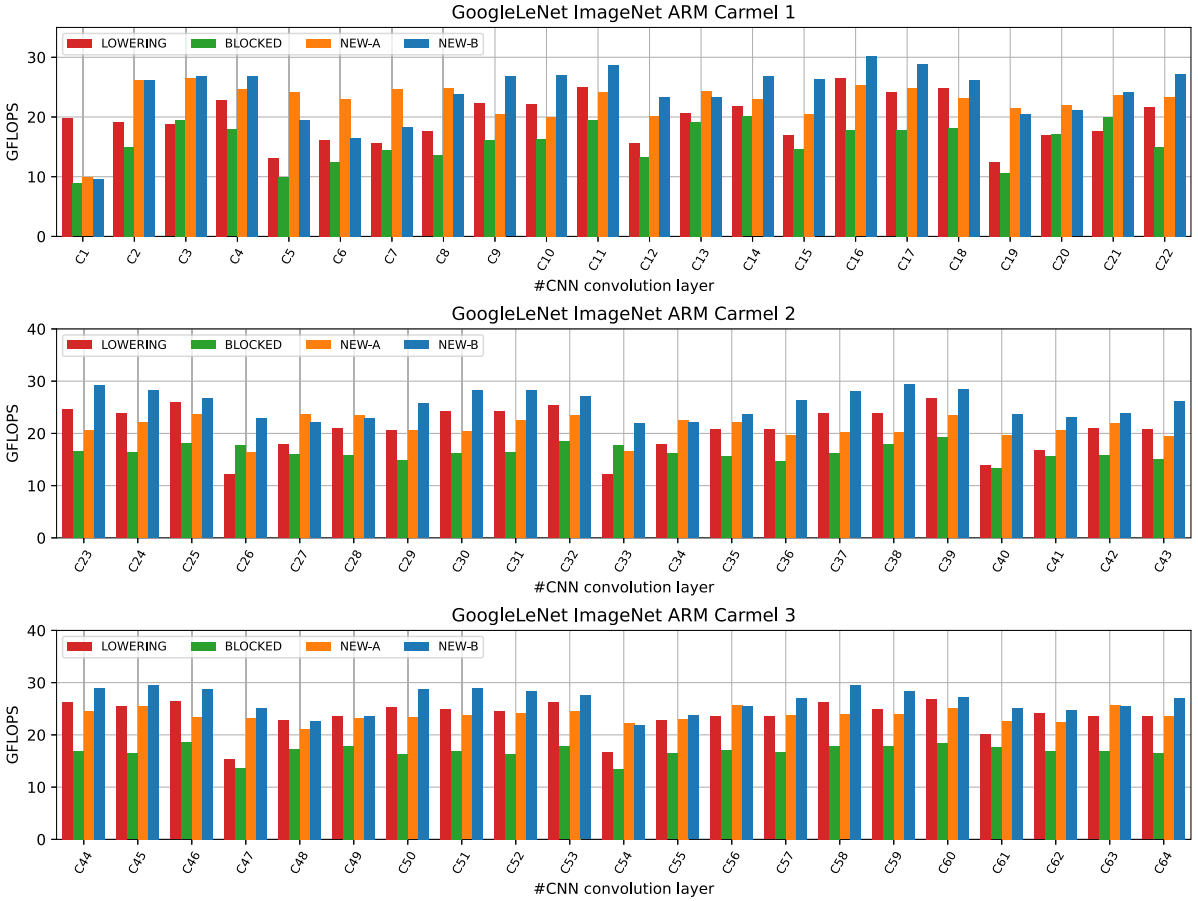

Fig. 10. Performance evaluation of the convolution operators for GoogleLeNet.

operator, many DL libraries implementing this operator either adopt the lowering approach or the less expensive, yet also less accurate and more "rigid", FFT and Winograd algorithms. In this paper, we refine and extend the original ideas in [7] to demonstrate that the direct convolution algorithm can be reformulated as a collection of loops around a microkernel that presents many similarities with the high performance, state-of-the-art implementations of GEMM. As a result, our new algorithms (1) preserve the conventional NHWC data layout for the layer input tensors; (2) (for the NEW-B variant) easily accommodate stride, dilation, and padding in the layer input tensor; (3) outperform the lowering approach by a fair margin; and (4) either require no memory overhead (NEW-A) at all or a very small workspace (for NEW-B). All in all, this work paves the road to considering the direct convolution as a first-class citizen, on par with other algorithms for this crucial operator.

As part of future work, we plan to analyze the extensions of the same ideas to the backward pass required in DNN training; the application

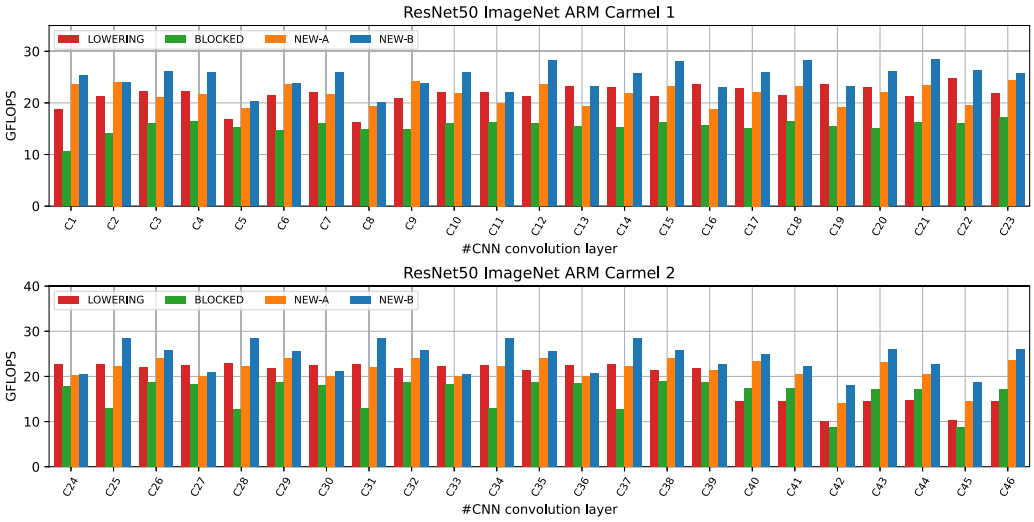

Fig. 11. Performance evaluation of the convolution operators for ResNet-50.

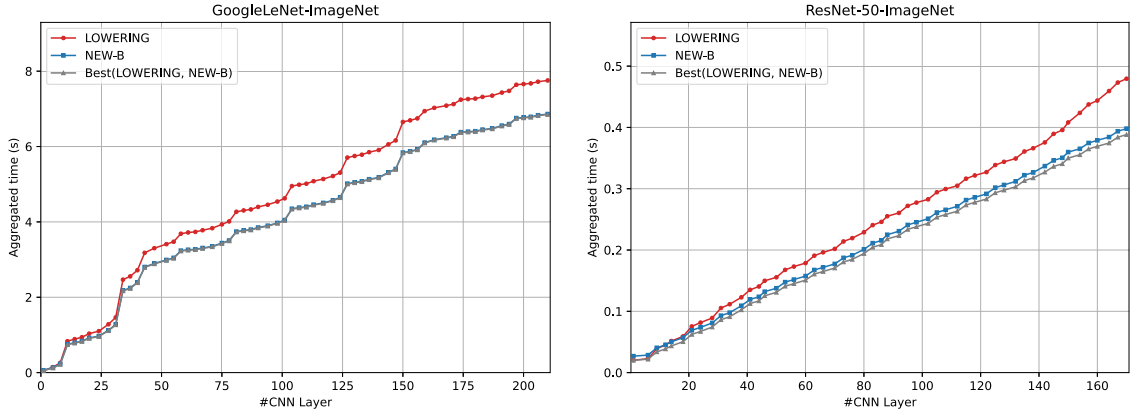

Fig. 12. Aggregated time of the convolutional layers for GoogleLeNet and ResNet-50.

of the discussed approach to the NCHW format; and its multithreaded parallelization using OpenMP.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

Acknowledgments

This work was supported by the research project PID2020-113656RB-C21/-C22 of MCIN/AEI/10.13039/501100011033. Adrián Castelló is a FJC2019-039222-I fellow supported by MCIN/AEI/10.13039/501100011033. Manuel F. Dolz was supported by the Plan Gen–T grant CDEIGENT/2018/014 of the *Generalitat Valenciana*. Héctor Martínez is a POSTDOC_21_00025 fellow supported by *Junta de Andalucía*. 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 European Union's Horizon 2020 research and innovation programme, and Spain, Germany, France, Italy, Poland, Switzerland, Norway.

References

[1] T. Ben-Nun, T. Hoefler, Demystifying parallel and distributed deep learning: An in-depth concurrency analysis, ACM Comput. Surv. 52 (4) (2019) 65:1–65:43, http://dx.doi.org/10.1145/3320060.

[2] V. Sze, Y.-H. Chen, T.-J. Yang, J.S. Emer, Efficient processing of deep neural networks: A tutorial and survey, Proc. IEEE 105 (12) (2017) 2295–2329, http://dx.doi.org/10.1109/JPROC.2017.2761740.

[3] K. Chellapilla, S. Puri, P. Simard, High performance convolutional neural networks for document processing, in: International Workshop on Frontiers in Handwriting Recognition, 2006.

[4] E. Georganas, S. Avancha, K. Banerjee, D. Kalamkar, G. Henry, H. Pabst, A. Heinecke, Anatomy of high-performance deep learning convolutions on SIMD architectures, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, SC '18, IEEE Press, 2018.

[5] P. San Juan, A. Castelló, M.F. Dolz, P. Alonso-Jordá, 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, http://dx.doi.org/10.1109/SBAC-PAD49847.2020.00023.

[6] Y. Zhang, Parallel Solution of Integral Equation-Based EM Problems in the Frequency Domain, IEEE Press, 2009.

[7] J. Zhang, F. Franchetti, T.M. Low, High performance zero-memory overhead direct convolutions, in: Proceedings of the 35th International Conference on Machine Learning – ICML, Vol. 80, 2018, pp. 5776–5785.

[8] W. Yang, J. Fang, D. Dong, X. Su, Z. Wang, LIBSHALOM: Optimizing small and irregular-shaped matrix multiplications on ARMv8 multi-cores, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '21, Association for Computing Machinery, New York, NY, USA, 2021, http://dx.doi.org/10.1145/3458817.3476217.

[9] A. Zlateski, Z. Jia, K. Li, F. Durand, The anatomy of efficient FFT and winograd convolutions on modern CPUs, in: Proceedings of the ACM International Conference on Supercomputing, ICS '19, Association for Computing Machinery, New York, NY, USA, 2019, pp. 414–424, http://dx.doi.org/10.1145/3330345.3330382.

[10] Q. Wang, D. Li, X. Huang, S. Shen, S. Mei, J. Liu, Optimizing FFT-based convolution on ARMv8 multi-core CPUs, in: M. Malawski, K. Rzadca (Eds.), Euro-Par 2020: Parallel Processing, Springer International Publishing, Cham, 2020, pp. 248–262.

[11] A. Zlateski, Z. Jia, K. Li, F. Durand, FFT convolutions are faster than Winograd on modern CPUs, here is why, 2018, arXiv preprint arXiv:1809.07851.

[12] A. Lavin, S. Gray, Fast algorithms for convolutional neural networks, in: 2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR, 2016, pp. 4013–4021, https://doi.ieeecomputersociety.org/10.1109/CVPR.2016.435.

[13] K. Goto, R.A. van de Geijn, Anatomy of high-performance matrix multiplication, ACM Trans. Math. Software 34 (3) (2008) 12:1–12:25.

[14] Z. Xianyi, W. Qian, Z. Yunquan, Model-driven level 3 BLAS performance optimization on Loongson 3A processor, in: 2012 IEEE 18th International Conference on Parallel and Distributed Systems (ICPADS), 2012.

[15] F.G. Van Zee, R.A. van de Geijn, BLIS: A framework for rapidly instantiating BLAS functionality, ACM Trans. Math. Softw. 41 (3) (2015) 14:1–14:33.

[16] X. Zhang, J. Xiao, G. Tan, I/O lower bounds for auto-tuning of convolutions in CNNs, in: Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '21, Association for Computing Machinery, New York, NY, USA, 2021, pp. 247–261, http://dx.doi.org/10.1145/3437801.3441609.

[17] G. Li, Z. Liu, F. Li, J. Cheng, Block convolution: Towards memory-efficient inference of large-scale CNNs on FPGA, 2021, http://dx.doi.org/10.48550/ARXIV.2105.08937.

[18] A. Zlateski, K. Lee, H.S. Seung, ZNN – a fast and scalable algorithm for training 3D convolutional networks on multi-core and many-core shared memory machines, in: 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS), IEEE, 2016, http://dx.doi.org/10.1109/ipdps.2016.119.

[19] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, M. Cowan, H. Shen, L. Wang, Y. Hu, L. Ceze, C. Guestrin, A. Krishnamurthy, TVM: An automated end-to-end optimizing compiler for deep learning, in: Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation, OSDI'18, USENIX Association, USA, 2018, pp. 579–594.

[20] H.-H. Liao, C.-L. Lee, J.-K. Lee, W.-C. Lai, M.-Y. Hung, C.-W. Huang, Support convolution of CNN with compression sparse matrix multiplication flow in TVM, in: 50th International Conference on Parallel Processing Workshop, ICPP Workshops '21, Association for Computing Machinery, New York, NY, USA, 2021, http://dx.doi.org/10.1145/3458744.3473352.

[21] E. Georganas, S. Avancha, K. Banerjee, D. Kalamkar, G. Henry, H. Pabst, A. Heinecke, Anatomy of high-performance deep learning convolutions on SIMD architectures, in: SC18: International Conference for High Performance Computing, Networking, Storage and Analysis, 2018, pp. 830–841, http://dx.doi.org/10.1109/SC.2018.00069.

[22] K. Dowd, C.R. Severance, High Performance Computing, second ed., O'Reilly, 1998.

[23] J.J. Dongarra, J. Du Croz, S. Hammarling, I. Duff, A set of level 3 basic linear algebra subprograms, ACM Trans. Math. Softw. 16 (1) (1990) 1–17.

[24] T.M. Low, F.D. Igual, T.M. Smith, E.S. Quintana-Ortí, Analytical modeling is enough for high-performance BLIS, ACM Trans. Math. Softw. 43 (2) (2016) 12:1–12:18.

[25] F.G. Van Zee, R.A. van de Geijn, The BLIS framework: Experiments in portability, ACM Trans. Math. Softw. 42 (2) (2016) 12:1–12:19.

[26] http://www.openblas.net, 2015.

[27] G. Alaejos, A. Castelló, P. Alonso-Jordá, F. Igual, E.S. Quintana-Ortí, Automatic generation of matrix multiplication routines for edge deep learning with TVM, IEEE Trans. Comput. (2022) In review.

[28] A. Castelló, S. Barrachina, M.F. Dolz, E.S. Quintana-Ortí, P.S. Juan, A.E. Tomás, High performance and energy efficient inference for deep learning on multicore ARM processors using general optimization techniques and BLIS, J. Syst. Archit. 125 (C) (2022) http://dx.doi.org/10.1016/j.sysarc.2022.102459.

[29] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S.E. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, A. Rabinovich, Going deeper with convolutions, 2014, CoRR abs/1409.4842, arXiv:1409.4842.

[30] K. He, X. Zhang, S. Ren, J. Sun, Deep residual learning for image recognition, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2016, pp. 770–778.

[31] A. Krizhevsky, I. Sutskever, G.E. Hinton, Imagenet classification with deep convolutional neural networks, in: Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1, NIPS'12, Curran Associates Inc., USA, 2012, pp. 1097–1105, http://dl.acm.org/citation.cfm?id=2999134.2999257.

[32] S. Barrachina, A. Castelló, M. Catalan, M.F. Dolz, J. Mestre, PyDTNN: a user-friendly and extensible framework for distributed deep learning, J. Supercomput. 77 (2021) http://dx.doi.org/10.1007/s11227-021-03673-z.

[33] S. Barrachina, A. Castelló, M. Catalán, M.F. Dolz, 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, http://dx.doi.org/10.1109/IPDPSW52791.2021.00110.