Parallel and Vectorised Winograd Convolutions for Multi-core Processors

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

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

Pedro Alonso-Jordá, Adrián Castelló,
Enrique S. Quintana-Ortí
Universitat Politècnica de València, Spain
palonso@dsic.upv.es,
{adcastel,quintana}@disca.upv.es

***Abstract*—We take a step forward toward developing high-performance codes for the convolution operator, based on the Winograd algorithm, on general-purpose processor architectures. In our approach, augmenting the portability of the solution is achieved via the introduction of vector instructions from Intel AVX2/AVX512 and ARM SVE to exploit the SIMD (single-instruction multiple-data) capabilities of current processors as well as OpenMP pragmas to exploit multi-threaded parallelism. While this comes at the cost of sacrificing a fraction of the computational performance, our experimental results on two distinct processors, with Intel Xeon Skylake and Fujitsu A64FX processors, show that the impact is affordable, and still renders a Winograd-based convolution that is competitive when compared with the lowering GEMM-based approach.**

***Index Terms*—Convolution, Winograd minimal filtering algorithm, high performance, vector intrinsics, SIMD units, multicore processors**

I. Introduction

Over the past years, convolutional neural networks (CNNs) have demonstrated excellent accuracy beyond their traditional application niches in computer vision and signal processing [1], [2]. The convolution operation in CNNs, though, is responsible for a major fraction of the computational cost required for training and inference [2]. It is therefore natural that a significant effort has been dedicated to developing and optimising efficient convolution algorithms.

Among the different methods for the convolution operator, we can list *i)* the direct algorithm; *ii)* the lowering (or IM2COL/IM2ROW-based) approach; *iii)* the FFT-based transform, and *iv)* the Winograd-based convolution. The corresponding high-performance implementations of some of these methods in libraries such as, for example, NVIDIA cuDNN and Intel oneAPI, is that the best performing and accurate method largely depends on the parameters that define the convolution.

In this work we address the efficient implementation of the Winograd-based convolution, using vector intrinsics, on current general-purpose processors equipped with SIMD FPUs (single-instruction multiple-data floating-point units). The use of vector intrinsics for this purpose, instead of "hand-coded" low-level assembly kernels (with vector instructions), in principle sacrifices some performance, but improves portability thanks to the support offered by current C compilers within platforms of a same vendor. In addition, the use of "high-level" codes with vector intrinsics eases the development of customised deep learning (DL) solutions via layer fusion. Specifically, we make the following contributions:

- We describe the implementation of the Winograd-based convolution enhanced with vector intrinsics for two processor architectures, Intel and ARMv8.2-SVE, using 256-bit, 512-bit and Vector-Length Agnostic (VLA) intrinsics, respectively defined in the Intel AVX2/AVX512 and ARM SVE.
- We present a "macro-tiling" technique that unrolls loops and fuses individual tiles of the Winograd algorithm to improve the utilisation of long vector registers as those present in Intel AVX2/AVX512 or ARM SVE intrinsics.
- We perform an evaluation of the implementations based on Intel AVX2/AVX512 and ARM SVE on two platforms equipped with Intel Xeon Gold and Fujitsu A64FX multicore processors. This experimental analysis includes our baseline Winograd-based convolution, an alternative lowering-based convolution algorithm, and two storage layouts, using two representative CNNs.

The rest of the paper is structured as follows. In Section II we briefly review the Winograd-based method for the convolution. Next, in Section III, we describe our "multi-platform" realisation of these algorithms with vector intrinsics and OpenMP using an Intel processor or a Fujitsu A64FX processor. In Section IV we evaluate the performance of the implementations on both target architectures, and finally, in Section V, we close the paper with a few concluding remarks.

II. Convolution Operators via the Winograd Minimal Filtering Algorithm

The Winograd (minimal filtering) algorithm provides a method to obtain an efficient realisation of a convolution operator [3]. Concretely, given a convolution layer that applies a filter f to an input image d, consisting of c input channels, in order to produce an output y, with k channels, the Winograd-based convolution can be expressed as

$$y_{i_k} = A^T \left(\textstyle\sum_{i_c=1}^{c} \left(G f_{i_k,i_c} G^T \right) \odot \left(B^T d_{i_c} B \right) \right) A, \quad i_k = 1, 2, \ldots, k, \tag{1}$$

where G, B respectively denote the transformation matrices for the filter and input matrices; A is the inverse transformation matrix; f_{i_k,i_c} is the i_c-th channel of the i_k-th filter; d_{i_c} is the i_c-th channel of the input image; y_{i_k} is the i_k-th channel of the output; and $\odot$ denotes the Hadamard (or element-wise) multiplication [4].

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 $m \times m$ output tile, with $t = m + r - 1$. An $h_i \times w_i$ image is processed by partitioning it into $t \times t$ tiles, with an overlapping factor of $r - 1$ elements between neighbouring tiles, yielding $\lceil h_i/m \rceil \lceil w_i/m \rceil$ tiles per channel. In this algorithm, choosing a larger value for m thus reduces the number of arithmetic operations, unfortunately at the cost of introducing numerical instability in the computation [5]. For that reason, m is usually set to be small, with two popular cases being $F(m \times m, r \times r) = F(4 \times 4, 3 \times 3)$ and $F(2 \times 2, 3 \times 3)$.

According to Winograd's formula (1), the intermediate Hadamard products are summed over all c channels to produce the i_k-th output channel. However, by properly scattering each transformed tile of the filter and input along the $t \times t$ dimensions, on respective intermediate workspaces U and V, of sizes $t \times t \times k \times c$ and $t \times t \times c \times (\lceil h_i/m \rceil \lceil w_i/m \rceil)$, both the Hadamard products and the element-wise summations can be collapsed into $t \times t$ independent matrix-matrix multiplications (also know as a "batched" GEMM). Finally, the same coordinates of the resulting $t \times t$ matrices are gathered to form a new $t \times t$ tile which is next used to compute the inverse transform as a $m \times m$ tile on the output tensor.

The general workflow of the "batched" GEMM variant of the Winograd algorithm exposes the four major phases: 1) filter transform; 2) input transform; 3) "batched" GEMM; and 4) output inverse transform. In DL, the 3D input/output tensors are extended with a batch dimension n using either the NCHW or the NHWC layouts.

III. High-Performance Winograd Convolution using Vector Intrinsics and OpenMP

In this section, we discuss several high-performance implementations of the Winograd algorithm, vectorised using Intel AVX2/AVX-512 and ARM SVE intrinsics, and parallelised using OpenMP. The vectorisation efforts for the Winograd algorithm have been applied to phases 1, 2, and 4. For brevity, we describe only the work on phases 1 and 2, corresponding to the filter and input transforms. Phase 3 can be seamlessly vectorised using a high-performance implementation of GEMM, for example, as that available in libraries such as Intel MKL, Intel oneAPI, BLIS or OpenBLAS, depending on the target architecture. Finally, we target single-precision floating-point (FP32) arithmetic.

A. Intel AVX2/AVX-512 intrinsics

In this section, we describe the implementation details of the input transform (phase 2) of the Winograd algorithm using Intel AVX2/AVX-512 intrinsics. The vectorisation of the filter and output inverse transforms (phases 1 and 4) of the Winograd algorithm follows a similar approach to that for the input transform. It is worth mentioning that each Winograd variant (using a given (m, r) pair) requires a specific vectorised implementation for phases 1, 2, and 4. This is due to the distinct dimensions and sparsity patterns of the transformation matrices G, A and B on the $m+r-2$ polynomial interpolation points [4]. Depending on the intrinsics used, each implementation implies, among other details, replacing the data types with Intel AVX2 `__m256` or AVX-512 `__m512`.

Given that the Winograd algorithm should leverage small values of m and r, such as $F(2 \times 2, 3 \times 3)$, $F(4 \times 4, 3 \times 3)$, vectorising these phases for 256- or 512-bit vector registers requires unrolling the loops iterating over the input tiles to fully exploit such long vector registers. By doing so, we design a "macro-tiling" technique that can then process a horizontal (and optionally vertical) block of consecutive tiles of the input/output images (or a subset of filters) in a single iteration so that the macro-tile columns, stored in vector registers, exploit their full length.

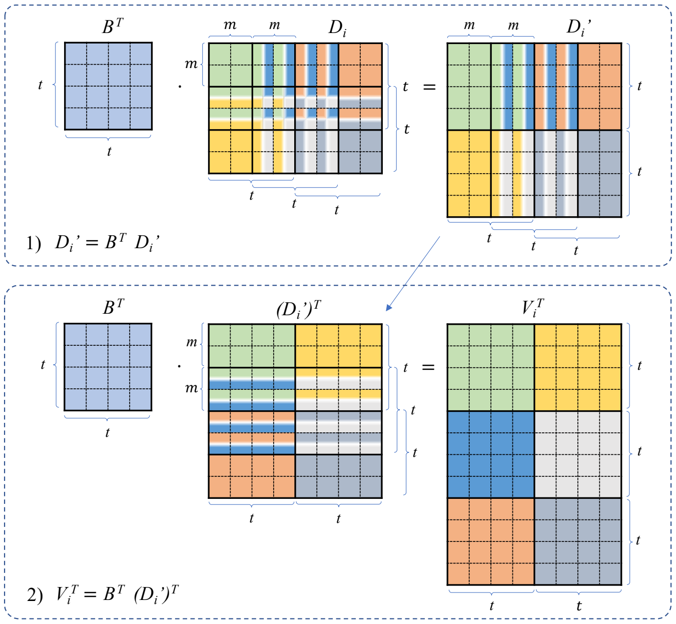

Fig. 1: Example of the Winograd macro-tiling technique for the input transform $V_i = B^T D_i B$ via the variant $F(2{\times}2, 3{\times}3)$ and using Intel AVX2 256-bit registers.

Figure 1 illustrates the macro-tiling technique for the input transform and the variant $F(2 \times 2, 3 \times 3)$ using Intel AVX2 256-bit SIMD units. In this code, the application of the input transform to a macro-tile is split into two sub-operations. The first performs the multiplication $D_i' = B^T \; D_i$, where D_i is a macro-tile of size $h_t \times w_t = 6 \times 8$, aggregating a 2×3 $d_{i,j}$ 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 can be reduced, given that the results of a tile can then be re-utilised for those that are immediately on the right. The second multiplication $V_i^T = B^T \; {D_i'}^T$ uses the previously transposed macro-tile D_i^T and is computed similarly, with the exception that there are no

```c
__m512  UX[10], WX[16];
t = 4;                     // Tile size
s = m = 2;                 // Tile stride (m)
timt_h= 4;  timt_w= 7; // Number of tiles per macro-tile
imt_h = t + (timt_h-1) * s;  // Input macro-tile height
imt_w = t + (timt_w-1) * s;  // Input macro-tile width
imt_vs= timt_h * s;  // Input macro-tile vertical stride
imt_hs= timt_w * s;  // Input macro-tile horiz. stride
// Number of vert./horiz. macro-tiles of the input image
imtile_h = ceil(((double)hi+2*vpadding-imt_h)/imt_vs)+1;
imtile_w = ceil(((double)wi+2*hpadding-imt_w)/imt_hs)+1;

#pragma omp parallel for \
   collapse(2) private(...) if ((n*c)>1)
for (in = 0; in < n; in++)
  for (ic = 0; ic < c; ic++)
    for (ih = 0; ih < imtile_h; ih++)
      // Calculate tile height bounds
      // (omitted for brevity)
      for (iw = 0; iw < imtile_w; iw++) {
        // Calculate tile width bounds ...
        // (omitted for brevity)
        // Set macro-tile to 0
        for (i = 0; i < imt_h; i++)
          UX[i] = _mm512_setzero();

        // Copy input to macro-tile
        for (i = fh; i < oh; i++)
          for (j = fw; j < ow; j++)
            UX[i][j] = Drow(in, ic, hh+i-fh, ww+j-fw);

        // WX  = Bt_row(i) * UX (rows of d)
        for (i = 0; i < timt_h; i++) {
          WX[i*4 + 0] =  UX[i*2 + 0] - UX[i*2 + 2];
          WX[i*4 + 1] =  UX[i*2 + 1] + UX[i*2 + 2];
          WX[i*4 + 2] = -UX[i*2 + 1] + UX[i*2 + 2];
          WX[i*4 + 3] =  UX[i*2 + 1] - UX[i*2 + 3];
        }
        // Transpose WX
        _MM_TRANSPOSE16_PS(WX[0], WX[1], WX[2], WX[3],
                           WX[4], WX[5], WX[6], WX[7],
                           WX[8], WX[9], WX[10],WX[11],
                           WX[12],WX[13],WX[14],WX[15]);

        // UX  = Bt_row(i) * WX
        int max_mth= min(tile_h-(ih*timt_h),timt_h),mth;
        int max_mtw= min(tile_w-(iw*timt_w),timt_w),mtw;

        for (mtw = 0; mtw < max_mtw; mtw++) {
          UX[0] =  WX[mtw*2 + 0] - WX[mtw*2 + 2];
          UX[1] =  WX[mtw*2 + 1] + WX[mtw*2 + 2];
          UX[2] = -WX[mtw*2 + 1] + WX[mtw*2 + 2];
          UX[3] =  WX[mtw*2 + 1] - WX[mtw*2 + 3];
          for (mth = 0; mth < max_mth; mth++)
            for (i = 0; i < t; i++)
              for (j = 0; j < t; j++)
                Vrow(i, j, ic, (in*tile_h*tile_w) +
                (iw*timt_w + mtw) + (ih*timt_h + mth) *
                tile_w) = UX[j][mth*t + i];
        }
      }
```

Listing 1: C code for the input transform vectorised using Intel AVX512 intrinsics the macro-tiling technique.

overlapped columns in the transposed resulting matrix V_i^T.

Listing 1 shows a excerpt of code for the Winograd variant $F(2 \times 2, 3 \times 3)$ for the input transform, implementing the macro-tiling technique using Intel AVX-512 intrinsics. In that code, the macro-tile of size 10×16 aggregates 4×7 input tiles of size 4×4. Furthermore:

- The "base vector datatype" corresponds to Intel AVX-512 `__m512`. The arrays of this type target the 512-bit ZMM vector registers with a capacity for 16 FP32 numbers.
- The input matrix is accessed via the C macro `DROW`, whose implementation is dependent on the type of storage layout being NCHW or NHWC.
- Loading the entries of the macro-tile is carried out via "scalar" operations. For the NCHW layout, this can be modified to take advantage of vector loads.
- The loop for the multiplication $D_i' = H^T \; D_i$. iterates over the vertical axis to uncollapse the $r-1$ overlapped rows in the resulting vectors `WX` which containing the macro-tile D_i' of size 16×16.
- The transposition of the matrix stored in the array `WX` is done via the C macro `_MM_TRANSPOSE16_PS`.
- The last nested loops perform the multiplication $V_i^T = B^T \; D_i'^T$. After processing a row of tiles within the macro-tile, the result stored in the four entries of the array `UX` is accordingly scattered across the entries of the workspace U.

As part of this work, we also vectorised the output transform using the macro-tiling technique. However, due to the more complex access pattern for the result tensor, this technique did not render any performance improvement. In consequence, our Intel AVX2- and AVX-512-based implementations only leverage the macro-tiling technique for the filter and input transform phases.

B. ARM SVE intrinsics

Compared to the AVX2/AVX-512 codes, implementing the filter transform using ARM SVE VLA intrinsics is not straightforward and requires rewriting the codes with some aspects in mind:

- It is not possible to declare SVE arrays with the vector datatype `svfloat32_t` since their size is not known at compile time. This forces us to rewrite and unroll some loops of the algorithm to mimic the behaviour of the SSE codes, resulting in more verbose implementations.
- The basic arithmetic operators are not overloaded by default, since the intrinsics require the use of masks (predicates), which have to be declared and initialised in advance. This further increases the code verbosity and reduces interpretability.

Listing 2 shows an excerpt of code for the filter transform phase using ARM SVE intrinsics. Compared to previous implementations, the code presents the following differences:

- The filter load is performed in a temporary variable (`F_tmp`) as the subscript operator (`[]`) is also not overloaded by default. Afterwards, the filter is loaded into the SVE registers via the intrinsic `svld1` with the predicate `pred3`, which was previously initialised via `svwhilelt_v32_u32` to operate only with the first 3 elements.
- The multiplication $W_{i_k} = G \; f_{i_k,i_c}$ is performed by steps, as the basic operators are not available for `svefloat32_t`. The same occurs for the multiplication $U_{i_k} = G \; W_{i_k}$.
- The transposition of the matrix stored in `W0–W3` is performed using a specialised in-house C macro implemented by the authors.
- The contents of registers `U0–U3` are stored via `svst1_f32`, with the `pred4` predicate, into the temporary matrix `U_tmp`. Finally, this matrix is copied to the corresponding entries of `U`.

In general, programming with SVE intrinsics has the advantage of generating a VLA code which does not need to

```c
svfloat32_t  F0, F1, F2,
             W0, W1, W2, W3,
             U0, U1, U2, U3;
svbool_t pred3 = svwhilelt_b32_u32(0, 3);
svbool_t pred4 = svwhilelt_b32_u32(0, 4);
int r = 3, t = 4;
float F_tmp[3][3], U_tmp[4][4];
// other declarations

#pragma omp parallel for \
   collapse(2) private (...) if ((k*c)>1)
for (ik = 0; ik < k; ik++)
  for (ic = 0; ic < c; ic++) {
    // U[..., ik, ic] = (G @ F[ik, ic, ...]) @ G.T

    // Load rows of 3x3 filter f
    for (i = 0; i < r; i++)
      for (j = 0; j < r; j++)
        F_tmp[i][j] = FROW(ik, ic, i, j);

    F0 = svld1(pred3, F_tmp[0]);
    F1 = svld1(pred3, F_tmp[1]);
    F2 = svld1(pred3, F_tmp[2]);

    // Wi   = G_row(i)   *  [ F0;F1;F2 ]
    W0 = F0;
    W1 = svadd_f32_z  (pred4, F0, F1);
    W1 = svadd_f32_z  (pred4, W1, F2);
    W1 = svmul_n_f32_z(pred4, W1, 0.5);
    W2 = svsub_f32_z  (pred4, F0, F1);
    W2 = svadd_f32_z  (pred4, W2, F2);
    W2 = svmul_n_f32_z(pred4, W2, 0.5);
    W3 = F2;

    // Transpose Wi
    SVE_TRANSPOSE4_F32(W0, W1, W2, W3);

    // Ui   = G_row(i) * [ W0;W1;W2;W3 ]
    U0 = W0;
    U1 = svadd_f32_z  (pred4, W0, W1);
    U1 = svadd_f32_z  (pred4, U1, W2);
    U1 = svmul_n_f32_z(pred4, U1, 0.5);
    U2 = svsub_f32_z  (pred4, W0, W1);
    U2 = svadd_f32_z  (pred4, U2, W2);
    U2 = svmul_n_f32_z(pred4, U2, 0.5);
    U3 = W2;

    // Scatter result in appropriate entries of U
    svst1_f32(pred4, U_tmp[0], U0);
    svst1_f32(pred4, U_tmp[1], U1);
    svst1_f32(pred4, U_tmp[2], U2);
    svst1_f32(pred4, U_tmp[3], U3);

    for (i = 0; i < t; i++)
      for (j = 0; j < t; j++)
        UROW(i, j, ik, ic) = U_tmp[j][i];
  }
```

Listing 2: C code for the filter transform vectorised using ARM SVE intrinsics.

be rewritten for other SVE architectures of different vector lengths. For the case of the Winograd algorithm, however, this does not bring major advantages as the vector length code strictly depends on the Winograd variant.

C. *Exploiting thread-level parallelism using OpenMP*

In addition to the introduction of vector intrinsics, the four phases of the algorithm can be also parallelised using OpenMP, as the individual kernels involved by the transform matrices for the filter/input/output tiles, as well as the $t \times t$ GEMM, present no data dependencies between them. To augment the degree of thread-level parallelism, we use the OpenMP `collapse(2)` clause to fuse the first two loops in each phase: across the k and c-dimensions in phase 1; the n and c-dimensions in phase 2; the two loops iterating over t in phase 3; and the n and k- dimensions in phase 4. This is shown, for example, in Listing 1 (Lines 13–14) for phase 2. Also, the $t \times t$ GEMM kernels in phase 3 are executed serially. Finally, we added the OpenMP clause `if` to extract thread-level parallelism only when the number of "collapsed" iterations is larger than 1.

IV. Experimental Results

In this section, we evaluate the performance of our implementations of the Winograd-based convolution on three different platforms, using two well-known DL models, for both the NCHW and NHWC data layouts. For comparison purposes, we also include the Lowering method (also referred to as IM2COL/IM2ROW + GEMM) [6], [7] in the evaluation. All experiments are performed using FP32 arithmetic.

A. *Hardware setup*

For the experiments, we employ the following platforms:

- SKY: This server comprises two Intel Xeon Gold (Skylake) 6126 processors (24 cores in total) running at 2.6 GHz and sharing 64 GiB of DDR4 RAM.
- A64FX: This is a Fujitsu PRIMEHPC FX1000 node equipped with a 48+4-core Fujitsu A64FX processor (ARMv8.2-SVE) running at 2.2 GHz. The cores of the node are grouped into four Core Memory Groups (CMGs), each with 12 compute cores plus an additional assistant core for the operating system. This machine contains 32 GiB of HBM2 memory.

In the experiments, we set the number of OpenMP threads to match the number of cores of a single socket for SKY (12), and the cores of a single CMG for A64FX (12 as well).

B. *DL framework, libraries, compilation flags, and parallelisation*

To evaluate our routines for the Winograd-based convolution, we bundled them into a dynamic C library and integrated the result with PyDTNN, a lightweight framework implemented in Python for DL training and inference [8], [9]. The compilation of this library is carried out using `gcc` v10.2.0 with the optimisation flags `-O3 -fopenmp` for both platforms in addition to `-mavx -mfma` for SKY. The $t \times t$ GEMM kernels in phase 3 are computed via Intel MKL v2022.1.0 (for SKY) and BLIS v0.8.1 (for A64FX).

Alternatively, the `Conv2D` layers in PyDTNN can be also processed via the IM2COL transform + GEMM of Lowering for the NCHW layout, or IM2ROW + GEMM for NHWC. The IM2COL/IM2ROW transforms are implemented in PyDTNN using Cython v0.29.24, and parallelised using OpenMP. The implementation of GEMM is provided by Numpy v1.23.0rc1, linked against Intel MKL for the SKY or BLIS for A64FX.

C. *Testbed*

For the evaluation, we measure the inference time spent by PyDTNN on the convolutional layers present in two popular DL models: VGG16 [10] and ResNet-50 (v1.5) [2] for the ImageNet dataset in both cases [11]. In our analysis, we only evaluate the convolutional layers using filters of size 3×3 with the Winograd variant $F(2 \times 2, 3 \times 3)$. For comparison purposes, we also measure the execution time of the Lowering approach using the same convolutional layers. The number of

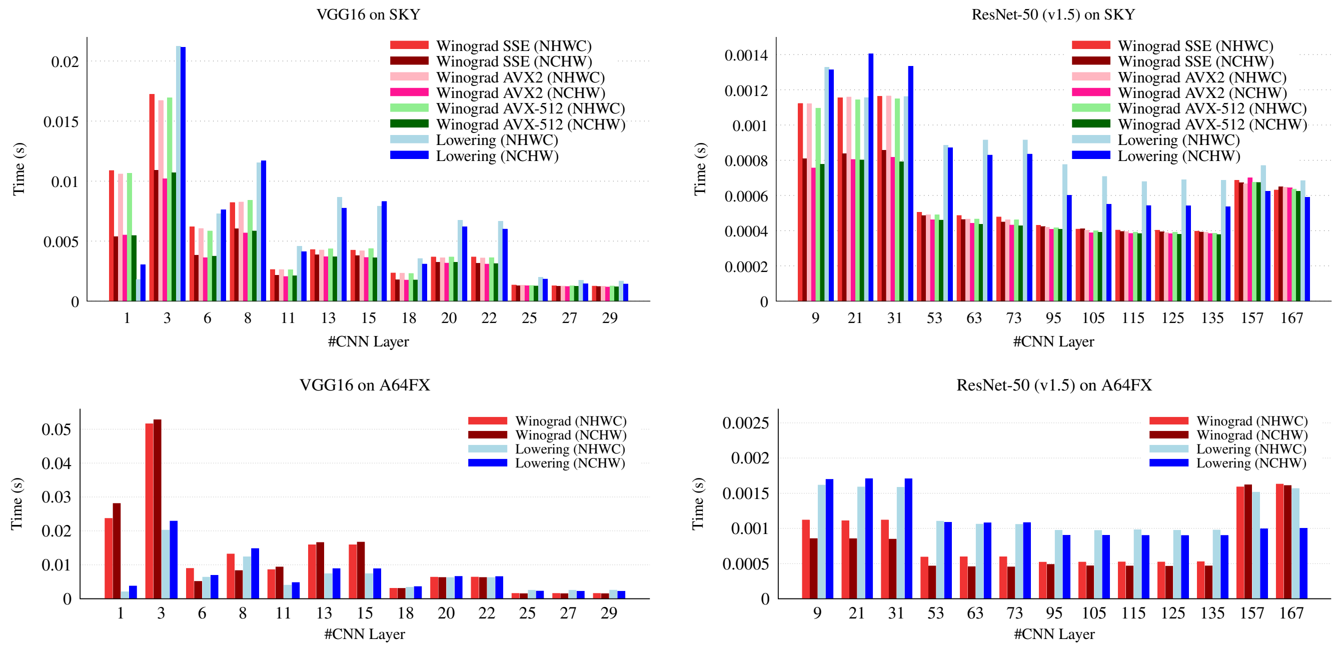

Fig. 2: Winograd vs Lowering execution time of the VGG16 (left) and ResNet-50 (right) convolution layers on the SKY (top) and A64FX (bottom) platforms using 12 threads.

images/batch size was set to $n = 1$ in order to reflect the single-stream scenario of the ML Commons benchmark. It is worth mentioning that the convolutional layer execution times in this analysis were averaged from 100 input images passed as inputs to the CNNs.

D. Results on the Intel Skylake

Figure 2 (top) reports the inference execution time of the convolutional layers using the Lowering method and the Winograd algorithm vectorised using AVX2 and AVX-512 intrinsics for both data layouts on the SKY platform. Note that in the experiments we also include the results using SSE intrinsics as baseline implementations with respect to the more sophisticated variants using AVX2/AVX-512 intrinsics.

The experiments show that, for VGG16, almost all convolutional operators appearing in the last layers of VGG16 (3–29), using either NCHW or NHWC and SSE, AVX2 or AVX-512 deliver higher performance than the Lowering approach. This is due to the reduction of the arithmetic cost implicit in the Winograd algorithm (potentially at the expense of less accurate results). Concerning the data layout we observe that, in general, NCHW offers higher performance than NHWC. This is due to the algorithm design, which first processes individual tiles for specific channels, accessing data contiguously according to NCHW. Regarding the use of SSE, AVX2 and AVX-512, we detect slightly smaller execution times for AVX2 and AVX-512, with no clear winner between them.

Focusing on ResNet-50, we can highlight similar results: the Winograd algorithm provides, for almost all convolutional layers (53–167), higher performance figures than the Lowering method. For the first layers (9–31), however, the Winograd algorithm using the NHWC format provides slightly less competitive results. This is due to the less efficient memory accesses performed by this algorithm for the NHWC data layout plus the convolutional parameters of these layers. We can also observe that, for the rest of the layers, the AVX2 and AVX-512 Winograd implementations deliver superior performance than SSE, with AVX2 providing slightly lower execution times than AVX-512.

E. Results on the Fujitsu A64FX

Figure 2 (bottom) reports the inference execution time of the convolutional layers using the Lowering method and the Winograd algorithm vectorised using ARM SVE intrinsics for both data layouts on the A64FX processor.

For VGG16, we observe that the Lowering method offers a significant advantage for the first three convolution layers of VGG16; however, for the remaining layers, the Winograd algorithm is a better option. For ResNet-50, the Winograd-based convolution is always the best option for both data layouts. In any case, the improvements of this algorithm mainly depend on the input and filter sizes.

We also detect differences between the two data layouts, with NCHW being more competitive than NHWC. These differences are due to the design of the Winograd algorithm, which first processes the individual tiles. A more efficient version of this algorithm would require a complete reformulation of the implementation to process first the tiles on the channel dimension, according to the storage of data in the NHWC layout.

V. Concluding Remarks

We have presented a collection of multi-threaded and vectorised implementations of the Winograd convolution operator via the use of 1) the OpenMP standard for the multithreaded parallelisation; 2) a reduced set of architecture-specific vector intrinsics (AVX2/AVX-512 for Intel and SVE for ARM); and 3) compiler support for high-level arithmetic operations involving vector registers.

The experimental results for two state-of-the-art platforms, equipped with SIMD-enabled Intel and Fujitsu multicore processors, show that our Winograd-based implementations, for the two most data layouts, deliver competitive performance compared with the Lowering approach on two of the platforms.

As future work, we plan to extend our study of vector intrinsics to other DL kernels, including the FFT convolution, as well as to target layer fusion and automatic generation of vectorised code to gain a more complete understanding of this procedure.

Acknowledgments

This research was funded by Project PID2020-113656RB-C21/C22 supported by MCIN/AEI/10.13039/501100011033. Manuel F. Dolz was also 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*. Adrián Castelló is a FJC2019-039222-I fellow supported by MCIN/AEI/ 10.13039/501100011033.

We thank the Barcelona Supercomputing Center for granting the access to the MareNostrum 4 CTE-ARM cluster based on Fujitsu FX1000 machines (with A64FX processors) where the developments and tests were performed.

References

[1] S. Pouyanfar, S. Sadiq, Y. Yan, H. Tian, Y. Tao, M. P. Reyes, M.-L. Shyu, S.-C. Chen, and S. S. Iyengar, "A survey on deep learning: Algorithms, techniques, and applications," *ACM Comput. Surv.*, vol. 51, no. 5, pp. 92:1–92:36, Sept. 2018. [Online]. Available: http://doi.acm.org/10.1145/3234150

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

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

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

[5] 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. [Online]. Available: https://doi.org/10.1145/3412380

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

[7] S. Barrachina, M. F. Dolz, P. San Juan, and E. S. Quintana-Ortí, "Efficient and portable GEMM-based convolution operators for deep neural network training on multicore processors," *J. Parallel Distrib. Comput.*, vol. 167, no. C, p. 240–254, sep 2022.

[8] 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.

[9] 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.

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

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