The Journal of Supercomputing (2023) 79:9819–9836
https://doi.org/10.1007/s11227-023-05050-4

Performance–energy trade-offs of deep learning convolution algorithms on ARM processors

Manuel F. Dolz[1] · Sergio Barrachina[1] · Héctor Martínez[2] · Adrián Castelló[4] · Antonio Maciá[3] · Germán Fabregat[1] · Andrés E. Tomás[4]

Accepted: 9 January 2023 / Published online: 21 January 2023
© The Author(s) 2023

Abstract
In this work, we assess the performance and energy efficiency of high-performance codes for the convolution operator, based on the direct, explicit/implicit lowering and Winograd algorithms used for deep learning (DL) inference on a series of ARM-based processor architectures. Specifically, we evaluate the NVIDIA Denver2 and Carmel processors, as well as the ARM Cortex-A57 and Cortex-A78AE CPUs as part of a recent set of NVIDIA Jetson platforms. The performance–energy evaluation is carried out using the ResNet-50 v1.5 convolutional neural network (CNN) on varying configurations of convolution algorithms, number of threads/cores, and operating frequencies on the tested processor cores. The results demonstrate that the best throughput is obtained on all platforms with the Winograd convolution operator running on all the cores at their highest frequency. However, if the goal is to reduce the energy footprint, there is no rule of thumb for the optimal configuration.

Keywords Convolution algorithms · ARM processors · High performance · Energy efficiency

1 Introduction

Over the past years, convolutional neural networks (CNNs) have exhibited an outstanding accuracy in a myriad of applications, including but not limited to face, speech, image or handwriting recognition, autonomous driving, and automatic medical diagnosis [1, 2]. The power and effectiveness of CNNs are due to the convolutional operators utilised therein, which can automatically detect distinctive image features while reducing the arithmetic costs and memory consumption required by previous approaches. This operator, however, is responsible for a substantial portion of the computational costs required for CNN training and

✉ Manuel F. Dolz
dolzm@uji.es

Extended author information available on the last page of the article

inference [2]. Consequently, significant efforts have been devoted to developing efficient algorithms for this particular computational core in almost all current processor architectures [3, 4]. In applications where CNNs inference is deployed on smart sensors or battery-operated devices equipped with micro-controller units (MCUs) (e.g. ARM Cortex-M CPUs) or low-power processors (e.g. ARM Cortex-A CPUs), the optimisation of this operator is strongly focused on reducing its energy consumption [5].

In this paper, we contribute to this line of work with a comprehensive analysis of the performance and energy efficiency of different convolution algorithms for deep learning (DL) inference on a collection of ARM-based processor architectures. Specifically, we make the following major contributions:

- We describe our high-performance implementations for the direct[1], explicit[2] and implicit lowering[3], and Winograd[4] convolution algorithms optimised for ARM processors and their integration within PyDTNN, a Python framework for training and inference of deep neural networks (DNNs) [6].
- We characterise the target ARM processors of different NVIDIA Jetson platforms based on Orin, Xavier, Nano, and TX2, as well as their execution models and the available power rails for measuring the CPU and memory energy consumption.
- We assess the performance and energy efficiency of the ResNet-50v1.5 CNN on varying configurations of convolution algorithms, number of threads/cores, ARM processors and operating frequencies. For each tested processor, we determine the best-performing configuration and the most energy-efficient scenario.

The rest of the paper is structured as follows. In Sect. 2, we describe some related works and compare their analysis with those obtained in this study. Next, in Section 3, we describe different convolution algorithms and optimisations for ARM processors. In Sect. 4, we evaluate the performance and energy consumption under different configurations, and finally, in Sect. 5, we close the paper with a few remarks and a brief discussion of future work.

2 Related work

Among different methods proposed in the literature for the convolution operator, we can list (i) the *direct algorithm*, usually implemented as six nested loops around a multiply-and-add instruction [4]; (ii) the *lowering (also known as* im2col/im2row

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

[2] The source code of the explicit lowering convolution is available at https://github.com/hpca-uji/PyDTNN.

[3] The source code of the implicit lowering convolution is available at https://github.com/hpca-uji/convGemm.

[4] The source code of the Winograd convolution algorithm is available at https://github.com/hpca-uji/convWinograd.

-based) approach, which transforms the input image(s) into a matrix in such a way that a general matrix–matrix (GEMM) multiplication can then be used to compute the convolution [7, 8]; (iii) the *FFT-based algorithm*, which shifts the computation into the frequency domain in order to reduce the arithmetic requirements [9–11]; and (iv) the *Winograd-based convolution*, which leverages the Winograd minimal filtering algorithm to decrease the arithmetic cost of the convolution [9, 12]. The general view of these methods and some of their corresponding high-performance implementations for MCUs and low-power processors (e.g. CMSIS-NN [13], ARM Compute Library [14], and NNPack [15]) is that the best option from the performance and energy viewpoints largely depends on the parameters that define the convolution operations (i.e. the dimensions of the filters and the image, the batch size, etc.), as well as different performance states and working modes provided by each processor architecture.

To perform DNN inference on low-power processors and MCUs, their limited computational and memory capabilities require aggressive optimisations in both algorithms and codes to be carried out. For instance, the authors in [16] propose a series of incremental improvements for DNN inference, such as the adjustment of the convolution algorithm or the cache blocking parameters, which are evaluated performance- and energy-wise on an NVIDIA Jetson XAVIER. Similarly, the authors in [17] present novel optimisation techniques based on layer separation and sparsification that are employed for wearable devices based on Qualcomm Snapdragon 400, ARM Cortex M0 and M3, and NVIDIA Tegra K1. Another approach for minimising resource requirements (computation, memory, and energy) is DeepX [18], a software tool that allows large-scale DL models to be executed efficiently on modern mobile processors. In this work, however, we focus on performing an exhaustive performance–energy evaluation of DL inference using a series of highly optimised convolution algorithms on ARM processors.

3 Convolution algorithms

The convolution operator

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

receives a sequence of filter tensors, F, and 4-dimensional (4D) inputs, I, to produce 4D output tensors, O, where:

- F comprises c_o filters of dimension $h_f \times w_f \times c_i$ each, where $h_f \times w_f$ correspond to the filters height $\times$ width.
- I consists of b input images of size $h_i \times w_i \times c_i$ each, where $h_i \times w_i$ denote the images height $\times$ width, and c_i stands for the number of input channels.
- O is composed of b outputs of size $h_o \times w_o \times c_o$ each, where $h_o \times w_o$ represent the outputs height $\times$ width and c_o is the number of output channels.

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 [2].

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

Listing 1 Basic algorithm for the direct convolution.

In the following sections, we review four different methods to compute the convolution operator and our high-performance implementations for ARM processors. In particular, we target: (i) a blocked variant of the direct algorithm (ConvDirect), (ii) a lowering approach (Lowering), (iii) an implicit lowering approach (ConvGemm), and (iv) a Winograd implementation (ConvWinograd).

3.1 Blocked algorithm for direct convolution

In previous work [19], we combined the blocking strategy in [4] for the direct convolution algorithm with the packing schemes employed in the high-performance formulation of GEMM [20]. The result was a new blocked version of the direct convolution, referred to as ConvDirect and illustrated by the algorithm in Listing 2, with the following properties:

- 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), mimicking the high-performance implementations of GEMM in GotoBLAS2, BLIS, OpenBLAS, and AMD AOML.
- The micro-tile dimensions are decoupled from the cache blocking parameters $w_{o,b}, c_{o,b}, c_{i,b}$.
- The input tensor contents 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 2 only shows where this packing routine is placed.)
- The filter tensor is re-packed into a 5D tensor, of dimension $h_f \times w_f \times c_o / c_{o,b} \times c_i \times c_{o,b}$. This type of packing enables unit-stride accesses to B from the micro-kernel. It should be noted that as the filters do not vary during inference, this only needs to be done once for the DNN model, and its cost becomes negligible.

A significant key to attaining high performance in the blocked direct convolution lies in the utilisation of an architecture-specific micro-kernel. The decoupling of the micro-tile dimensions from the cache blocking parameters combined with the packing of the input tensor facilitates leveraging existing high-performance micro-kernels, specifically tuned for a concrete processor architecture [19]. The advantage of our approach is to directly handle the well-adopted NHWC data layout, avoiding the tensor transformation overhead of previous algorithm designs [4].

```
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                // Packing of A_c (omitted for simplicity)
10               for ( j' = 0; j' < c_o/c_{o,b}; j'++ )
11                 for ( jj = 0; jj < c_{o,b}; jj += n_r )
12                   for ( kk = 0; kk < w_{o,b}; kk += m_r ) {
13                     // Micro-kernel
14                     for ( ii = 0; ii < c_{i,b}; ii++ )
15                       for ( j_r = kk; j_r < kk + m_r; j_r++ )
16                         for ( i_r = c_{o,b}; i_r < c_{o,b} + n_r; i_r++ )
17                           O[h][l][k' · w_{o,b} + j_r][j' · c_{o,b} + i_r]
18                             += I[h][l + n][k' · w_{o,b} + j_r + m][i' · c_{i,b} + ii]
19                              · F[j' · c_{o,b} + i_r][n][m][i' · c_{i,b} + ii];
20 }}}
```

Listing 2 Blocked variant of the direct convolution.

Micro-kernels for the blocked direct convolution The evaluated direct convolution framework includes a number of micro-kernels specifically developed and tuned for the ARM Neon v8. These micro-kernels work with different micro-tile sizes, MR and NR, like 8×12, 4×12, 4×16, 4×20, etc. Among these, the 8×12 micro-kernel is the one which generally attains the best performance. For this reason, we have selected this micro-kernel size for the performed ConvDirect experiments. It should be noted that this micro-kernel uses two different implementations. The first one is used when the micro-tile size is equal to MR × NR (8×12) and is programmed in ARM assembly language. The second implementation, which is called in those cases where the size of the micro-tile is smaller than MR × NR, is programmed using ARM intrinsic instructions.

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. In our case, we parallelised the loop traversing the $c_{o,b}$ dimension (see Line 11 of Listing 2).

3.2 Lowering approach

A high-performance implementation of the convolution operator can be obtained for current computer architectures by lowering this operator into 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. Applies a row transformation to the 4D input tensor I in order to build an augmented 2D matrix A of size $m \times k = (bh_ow_o) \times (c_ih_fw_f)$ [7], as shown in the algorithm in Listing 3.
2. Computes the output of the convolution directly from the GEMM $C = A \cdot B$, 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.

We denote the combination of these two steps as an explicit IM2ROW-based convolution. The lowering approach performs 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    for ( h = 0; h < b; h++ )
3      for ( i = 0; i < c_i; i++ )
4        for ( k = 0; k < w_o; k++ )
5          for ( l = 0; l < h_o; l++ ) {
6            r = l + k h_i + h w_i h_i;
7            for ( m = 0; m < w_f; m++ )
8              for ( n = 0; n < h_f; n++ ) {
9                c = n + m h_f + i w_f h_f;
10               A[r][c] = I[h][l + n][k + m][i];
11 }}}
```

Listing 3 Algorithm for the IM2ROW transform.

Parallelising and vectorising the IM2ROW transform. The IM2ROW transform is a memory-bound transform which can be easily parallelised by adding the appropriate OpenMP directive to the most appropriate loop(s), preferably one of the outermost ones or a collapsed combination of them. In our case, we parallelised loops iterating over the batch size and input channels, though the parallelisation efficiency is limited when the multiple threads saturate the memory bandwidth while accessing to I and A or when the number of collapsed iterations is not big enough.

Its vectorisation, however, is not straightforward. This kernel only performs data movements between I and the workspace A, so we efficiently benefit from SIMD loads (e.g. via ARM NEON intrinsics) provided the innermost loop traverses the c_i dimension of the problem. However, SIMD stores are not possible to write the values to A.

High-performance GEMM and lowering. On the positive side, the large dimension of the GEMM appearing in the lowering approach favours an efficient

vectorisation and exposes a high degree of loop-level parallelism for multicore ARM architectures. This can be achieved by invoking an existing high-performance implementation of GEMM, such as that in the BLIS framework [20].

3.3 Implicit lowering approach

The negative side of the lowering approach described in the previous subsection is that it requires a large temporary workspace A ($h_f w_f$ times larger than I) and presents a certain overhead due to the required data copies. To minimise these costs, instead of explicitly constructing the matrix A, it is possible to combine the IM2ROW transform and the packing of A into the buffer A_c that occurs within the GEMM BLIS [3]. For this purpose, during the execution of the GEMM kernel, the buffer A_c is directly constructed from the contents of the input tensor I, instead of from the augmented matrix A. This approach never explicitly assembles A, obtaining large memory savings because A_c typically requires a few kilobytes of memory and is much smaller than A. The maximum size of A depends on the model, the data set, the batch size, and the data type being used. For example, for ResNet50, ImageNet, a batch size of 1, and using the float32 data type, the maximum size of A is 6.9 MiB, whereas with a batch size of 64 is 441 MiB. Our implementation for the implicit lowering convolution variant is denoted as ConvGemm.

3.4 Winograd minimal filtering algorithm

The Winograd (minimal filtering) [21] algorithm, referred to as ConvWinograd, provides a method to obtain an efficient implementation of a convolution operator [22]. 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

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

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 [12].

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 [23]. 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 (2), the intermediate Hadamard products are summed over all c channels to produce the i_k-th output channel. In our Winograd implementation [24], we scatter 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 $t \times t \times c \times (\lceil h_i/m \rceil \lceil w_i/m \rceil)$ in order to collapse the Hadamard products and the element-wise summations into $t \times t$ independent matrix–matrix multiplications (also known as a *batched* GEMM [25]). 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.

In summary, the batched GEMM variant of the Winograd algorithm exposes 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.

OpenMP parallelisation In our implementation, the four phases of the algorithm are 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 also use the OpenMP `collapse` clause to fuse the first two loops in each phase. Each individual $t \times t$ GEMM kernel in phase 3 is executed serially, but we parallelise their calculation across the $t \times t$ dimensions.

Vectorising the input transform The implementation of the Winograd filter/input/output transform phases is also vectorised using ARM NEON intrinsics. A specialised GEMM for the filter/input/output tiles is implemented computing only the nonzero elements in the G, B, and A sparse transformation matrices. Given that these matrix operands remain static for all the computation, it is possible to hard-code their entries to directly operate with vector registers.

4 Experimental results

In this section, we assess the performance and energy efficiency of the described convolution algorithms on four different platforms based on ARM multicore CPUs with varying core/frequency configurations using the ResNet-50 v1.5 CNN model. In [16], we performed an exhaustive execution analysis of different algorithms and described a series of optimisations to reduce the inference time. In this paper, we leverage the optimal versions of these algorithms.

4.1 Hardware setup

For the experiments, we use a range of NVIDIA Tegra platforms [26], a SoC series for battery-operated devices such as smartphones, personal digital assistants, and mobile Internet devices. The Tegra SoCs integrate an ARM architecture CPU, a GPU, a north/southbridge, and a memory controller into a single package. These low-power SoCs, branded as NVIDIA Jetson, emphasise performance for gaming and machine/deep learning applications with a special focus on energy efficiency.

Table 1 Specifications for the selected NVIDIA Jetson platforms

Platform	Component	Description
Orin	CPU	12×ARM Cortex-A78AE ARMv8.2 cores @ 2.20 GHz (grouped in 3 clusters)
	L1 Cache	64 KiB ICache; 64 KiB DCache (per core)
	L2 Cache	256 KiB (per core)
	L3 Cache	2048 KiB (per cluster)
	Memory	64 GiB LPDDR5 @ 204.8 GB/s
Xavier	CPU	8×NVIDIA Carmel ARMv8.2 cores @ 2.26 GHz (grouped in 4 clusters)
	L1 cache	128 KiB ICache; 64 KiB DCache (per core)
	L2 cache	2048 KiB (per cluster)
	L3 cache	4096 KiB (total)
	Memory	32 GiB LPDDR4 @ 137 GB/s
Nano	CPU	4×ARM Cortex-A57 ARMv8-A cores @ 1.43 GHz
	L1 cache	48 KiB ICache; 32 KiB DCache (per core)
	L2 cache	2048 KiB (total)
	Memory	4 GiB LPDDR4 @ 25.6 GB/s
TX2	CPU	2×NVIDIA Denver2 ARMv8-A cores @ 2.0 GHz
		4×ARM Cortex-A57 ARMv8-A cores @ 2.0 GHz
	L1 cache	Denver: 128 KiB ICache; 64 KiB DCache (per core)
		ARM: 48 KiB ICache; 32 KiB DCache (per core)
	L2 cache	2048 KiB (per cluster)
	Memory	8 GiB LPDDR4 @ 59.7 GB/s

Table 2 Selected NVP model configurations for the NVIDIA Jetson platforms

Platform	NVP model ID	NVP model	CPUs enabled	Power budget (W)	Frequency (GHz)
Orin	0	Max-N	12	–	2.20
	1	15W	4	15	1.11
	2	30W	8	30	1.72
	3	50W	12	50	1.49
Xavier	0	Max-N	8	–	2.26
	1	10W	2	10	1.20
	2	15W	4	15	1.20
	3	30W-All	8	30	1.20
Nano	0	Max-N	4	–	1.48
	1	5W	2	5	0.92
TX2	0	Max-N	2×Denver2 + 4×A57	–	2.00
	1	Max-Q	4×A57	–	1.20
	2	Max-P-Core-All	2×Denver2 + 4×A57	–	1.40
	3	Max-P-ARM	4×A57	–	2.00

Table 3 Measured CPU-related power rails for each system platform

Platform	Measured power rails	Description
ORIN	+VDD_CPU_CV	CPU and CV combined power rail
	+VDDQ_VDD2_1V8AO	DDR and 1V8AO combined power rail
XAVIER	+CPU	CPU power rail
	+VDDRQ	DDR memory power rail
NANO	+POM_5V_IN	System 5V power rail
	– POM_5V_GPU	GPU power rail (subtracted from POM_5V_IN)
TX2	+VDD_SYS_CPU	CPU power rail.
	+VDD_SYS_DDR	DDR memory power rail

Table 1 describes the specifications of the four selected NVIDIA Jetson platforms: AGX ORIN, AGX XAVIER, NANO, and TX2. For each platform, the table details the CPU model, architecture, maximum operating frequency, levels of cache, memory type and size. Each of these systems permits the selection of different performance models, also known as NVP (NVIDIA Performance) models, allowing the user to enable/disable CPUs and to adjust the core operating frequency in order to limit the maximum power consumption. NVP models can be configured and monitored via the nvpmodel and jetson-stats tools [27], respectively. The performance models configured for each platform are detailed in Table 2. There, the Max-N mode enables all CPU cores and allows them to consume as much power as they require to achieve their maximum performance. In contrast, modes with IDs starting from 1 on (except for TX2) set a maximum power budget to be consumed, either by progressively disabling cores, reducing the operating frequency, or a combination of both. Note that the TX2 platform does not offer any kind of power capping in its available NVP models.

4.2 Power monitoring

The Jetson platforms offer several INA3221 on-board power sensors via I^2C, which can be monitored through the sysfs file system nodes [28]. These sensors measure power, voltage, and current rails available for each platform. Table 3 details the CPU-related rails that we monitor for the energy efficiency evaluation. These include the CPU and memory (DDR) power consumption rails but omit those related to the GPU and other on-board peripherals that have not been used in this study.

To collect measurements from these power rails, we leverage PMLIB, a Power Measurement Library [29]. This library implements a client-server model where the server continuously reads power samples that can be later collected by the clients for a given time interval. To gather power measurements from each Jetson platform, the corresponding PMLIB module reads the sysfs files related to the power rails listed in Table 3 at a frequency of 10 Hz.

4.3 DL framework, libraries, compilation flags, and parallelisation

To evaluate different convolution algorithms, we bundled their codes into individual C libraries and integrated them into PyDTNN, a lightweight framework implemented in Python for DL training and inference [6, 30]. For this purpose, we developed the corresponding binding modules that internally call the ConvDirect, ConvGemm, and ConvWinograd C functions via the `ctypes` Python library. These Python modules interact with the PyDTNN layer class `Conv2D` to finally execute the convolution algorithm. For the case of ConvWinograd, the binding module calls the Winograd C function according to the filter size requested by the convolutional layer encountered in a given neural network model. Note that the libraries for the ConvGemm and ConvWinograd algorithms, which internally execute the GEMM or leverage a GEMM-related micro-kernel, are linked against BLIS v0.8.1. Alternatively, the implementation of the Lowering algorithm within the `Conv2D` layers is implemented in PyDTNN using Cython v0.29.24 and parallelised with OpenMP. In this case, the implementation of GEMM is provided by the same BLIS library. It is also worth noting that the three convolution libraries (ConvGemm, ConvDirect and ConvWinograd) have been compiled using `gcc` v10.2.0 with the optimisation flags `-O3 -fopenmp` for all the platforms.

4.4 Testbed

For the evaluation, we measure the inference throughput and energy efficiency in terms of images per second and images per Joule, respectively, of the ResNet-50 v1.5 [2] CNN model on the ImageNet dataset [31]. In all cases, the batch size is set to $n = 1$ to reflect the single-stream scenario of the ML Commons benchmark for inference on edge computing. Also, all the operations performed in the CNN inference experiments are carried out using FP32 arithmetic.

4.5 Performance and energy efficiency scalability

Figure 1 reports the inference throughput (left-hand column) and energy efficiency (right-hand column) for the four NVIDIA Jetson platforms with a varying number of threads and NVP models. Note that both throughput and energy efficiency figures were averaged from a total of 600 inferences using the ResNet-50 v1.5 model. From the performance point of view, we observe that increasing the operating frequency and the number of threads delivers the best throughput in general. This behaviour is common for all the platforms and algorithms, except for the ConvDirect on the XAVIER, where the scalability is limited by the improper use of cache memories in a multithreaded scenario.[5] Leaving apart this outlying result, the algorithm

[5] We suspect that the ConvDirect algorithm is not properly considering the cache replacement policy on this platform, though we leave its optimisation as part of a future work.

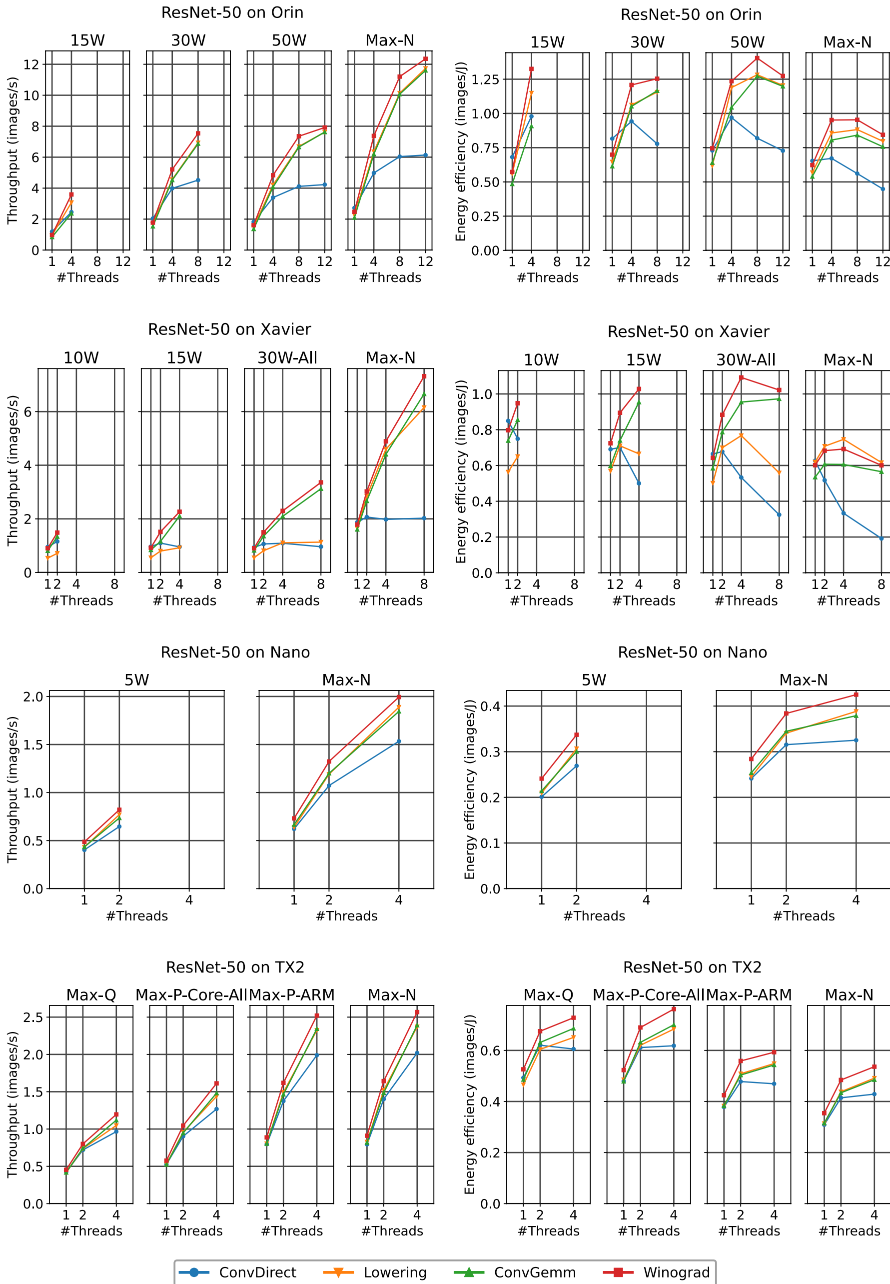

Fig. 1 Performance and energy efficiency per convolution algorithm, number of threads and NVP model

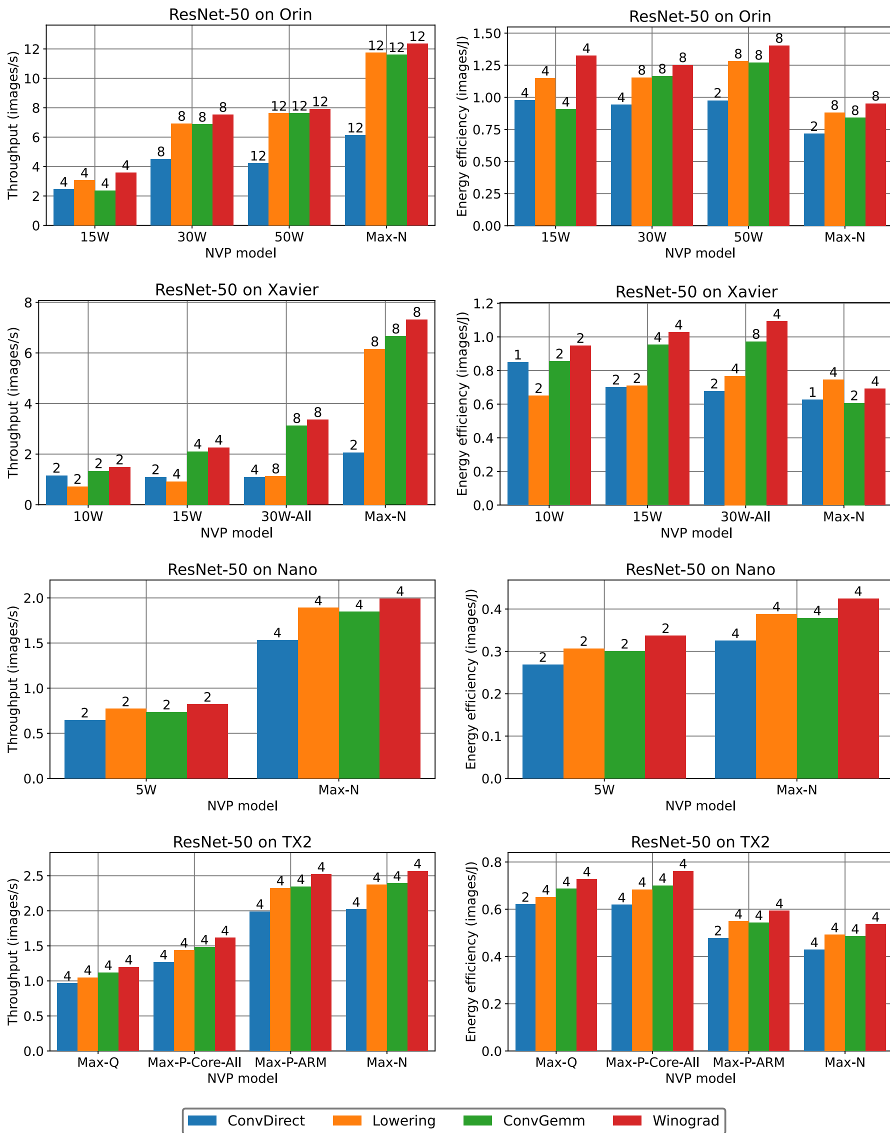

Fig. 2 Performance and energy efficiency with the best thread configuration for each algorithm and NVP model

that delivers in general the best performance in all configurations and platforms is ConvWinograd.

Focusing on energy efficiency, we observe different trends depending on the selected NVP model, number of threads, and platform. The first observation is that the best energy efficiency is not always obtained by increasing the number of threads. The most energy-efficient configuration for Orin on the 50W model

Table 4 Optimal thread and NVP model configurations for best performance and energy efficiency

Platform	Method	Configuration for best performance			Configuration for best energy efficiency		
		NVP mode/#Cores	Images/s	Images/J	NVP mode/#Cores	Images/s	Images/J
Orin	ConvDirect	Max-N/12	6.14	0.45	15W/4	2.45	0.98
	Lowering	Max-N/12	11.73	0.80	50W/8	6.69	1.28
	ConvGemm	Max-N/12	11.61	0.76	50W/8	6.65	1.27
	ConvWino-grad	Max-N/12	**12.35**	0.84	50W/8	7.35	**1.40**
Xavier	ConvDirect	Max-N/2	2.07	0.52	10W/1	0.95	0.85
	Lowering	Max-N/8	6.15	0.62	30W-All/4	1.11	0.77
	ConvGemm	Max-N/8	6.66	0.57	30W-All/8	3.12	0.97
	ConvWino-grad	Max-N/8	**7.32**	0.60	30W-All/4	2.30	**1.09**
Nano	ConvDirect	Max-N/4	1.53	0.33	Max-N/4	1.53	0.33
	Lowering	Max-N/4	1.89	0.39	Max-N/4	1.89	0.39
	ConvGemm	Max-N/4	1.84	0.38	Max-N/4	1.84	0.38
	ConvWino-grad	Max-N/4	**1.99**	0.42	Max-N/4	1.99	**0.42**
TX2	ConvDirect	Max-N/4	2.02	0.43	Max-Q/2	0.72	0.62
	Lowering	Max-N/4	2.37	0.49	Max-P-Core-All/4	1.44	0.68
	ConvGemm	Max-N/4	2.39	0.49	Max-P-Core-All/4	1.48	0.70
	ConvWino-grad	Max-N/4	**2.57**	0.54	Max-P-Core-All/4	1.61	**0.76**

is to use only 8 threads of 12 for all algorithms except ConvDirect. For Xavier on the 30W-All model, the best choice is to set 4 cores instead of all 8. Similar to the previous performance study, the ConvDirect algorithm on Xavier and Orin obtains a lower energy efficiency as the number of threads increases. Again, this is due to the poor usage of the cache levels that the configuration of this algorithm delivers on these specific platforms. On the contrary, the increasing number of threads on the Nano and TX2 platforms does improve their energy efficiency. Therefore, we can conclude that these platforms are far more energy proportional than Orin and Xavier, as the energy consumed by the algorithm decreases when more cores (threads) are used.

4.6 Performance and energy consumption for the best parallel configuration

Figure 2 summarises the results shown in Figure 1 by reporting only those configurations with the number of threads that deliver the best throughput (left-hand

column) and energy efficiency (right-hand column) per algorithm and NVP model. The numbers at the top of the bars indicate the optimal number of threads of each configuration. Focusing on performance, we can conclude, as already mentioned above, that increasing the NVP model (from model ID 1 on) in each device improves the performance of different algorithms. Besides, the algorithm with the best throughput is ConvWinograd with the Max-N model. This combination of algorithm and NVP model outperforms all other configurations, except on TX2 where the Max-P-ARM model obtains a behaviour similar to Max-N.

Concerning energy efficiency, the results are less predictable. Nevertheless, the most energy-efficient algorithm is still ConvWinograd. Instead, the best NVP model for each platform is 50W for Orin, 30W-All for Xavier, Max-N for Nano, and Max-P-Core-All for TX2.

Table 4 reports the NVP model and the number of threads (cores) tuple configurations that achieve the best throughput and energy efficiency obtained by each convolution algorithm. The throughput and energy efficiency of the best algorithm for each platform are highlighted in bold.

All in all, we can conclude that if the aim is to maximise performance, the best option on all the platforms is to use the ConvWinograd algorithm and set the NVP model to Max-N with the maximum number of cores. In contrast, if the goal is to reduce the energy footprint, there is no rule of thumb, as each platform has its own optimal configuration.

5 Concluding remarks

We have performed an exhaustive evaluation of the performance and energy efficiency attained by the selected convolution algorithms for DL inference on a collection of low-power ARM-based processor architectures. In particular, this analysis leverages a collection of NVIDIA Jetson platforms based on Orin, Xavier, Nano, and TX2, comprising different models of ARMv8-A-/ARMv8.2-based multicore processors.

Regarding the experimental results, we can conclude that for raw DL inference performance, the most suitable configuration consists of scaling the processors to the highest available frequency and using the maximum number of cores. However, this configuration will only achieve great results provided the convolution algorithm delivers fair strong scaling and makes efficient use of the cache levels. In contrast, if the goal consists in reducing the energy footprint, we find no clear winner for an optimal configuration, as depending on the platform, algorithm, the number of cores, and operating frequency, we can find different outcomes. We relate these effects to the distinct energy efficiencies and energy proportionality grades of the assessed Tegra SoC devices for the evaluated convolution algorithms.

As part of the future work, we plan to complement the performance–energy trade-off analysis with high-end processors, such as the Fujitsu A64FX ARMv8.2-A+SVE, and MCUs, equipped with ARM Cortex-M and ESP32 CPUs, e.g. the Arduino Nano 33 BLE Sense or the Espressif ESP32-EYE devices. To reproduce

this experimentation on those platforms, we plan to develop external power–energy meters providing high-resolution measurements.

Acknowledgements Not applicable.

Author Contributions All authors contributed on the definition of the evaluated convolution algorithms, on how to integrate them into PyDTNN and on how to correctly assess their performance and energy consumption on different testing platforms. The code implementation and the manuscript writing were led by Manuel F. Dolz and Héctor Martínez. The preparation and execution of all the experiments on the computing platforms were led by Sergio Barrachina. Finally, all authors revised and improved different versions of the manuscript.

Funding Open Access funding provided thanks to the CRUE-CSIC agreement with Springer Nature. 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. Antonio Maciá is a PRE2021-099284 fellow supported by MCIN/AEI/10.13039/501100011033.

Availability of data and materials The ImageNet dataset used for the current study is publicly available from the web. See https://www.image-net.org/.

Declarations

Ethics approval and consent to participate Not applicable.

Consent for publication Not applicable.

Conflict of interest The authors declare that they have no competing interests.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Pouyanfar S, Sadiq S, Yan Y, Tian H, Tao Y, Reyes MP, Shyu M-L, Chen S-C, Iyengar SS (2018) A survey on deep learning: Algorithms, techniques, and applications. ACM Comput Surv 51(5):92:1-92:36. https://doi.org/10.1145/3234150. (**[Online]**)
2. Sze V, Chen Y-H, Yang T-J, Emer JS (2017) Efficient processing of deep neural networks: a tutorial and survey. Proc IEEE 105(12):2295–2329
3. San Juan P, Castelló A, Dolz MF, Alonso-Jordá P, Quintana-Ortí ES (2020) High performance and portable convolution operators for multicore processors. In: 2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), pp 91–98
4. Zhang J, Franchetti F, Low TM (2018) High performance zero-memory overhead direct convolutions. In: Proceedings of the 35th International Conference on Machine Learning – ICML, Vol 80, pp 5776–5785
5. Pantho MJH, Bhowmik P, Bobda C (2021) Towards an efficient CNN inference architecture enabling in-sensor processing. Sensors 21(6):1955

6. Barrachina S, Castelló A, Catalan M, Dolz MF, Mestre J (2021) PyDTNN: a user-friendly and extensible framework for distributed deep learning. J Supercomput 77:09
7. Chellapilla K, Puri S, Simard P (2006) High performance convolutional neural networks for document processing. In: International Workshop on Frontiers in Handwriting Recognition
8. Georganas E, Avancha S, Banerjee K, Kalamkar D, Henry G, Pabst H, Heinecke A (2018) 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
9. Zlateski A, Jia Z, Li K, Durand F (2019) The anatomy of efficient FFT and Winograd convolutions on modern CPUs. In: Proceedings of the ACM International Conference on Supercomputing, ser. ICS '19. New York, NY, USA: Association for Computing Machinery, p 414-424. Available: https://doi.org/10.1145/3330345.3330382
10. Wang Q, Li D, Huang X, Shen S, Mei S, Liu J (2020) Optimizing FFT-based convolution on ARMv8 multi-core CPUs. In: Malawski M, Rzadca K (eds) Euro-Par 2020: Parallel Processing. Springer International Publishing, Cham, pp 248–262
11. Zlateski A, Jia Z, Li K, Durand F (2018) FFT convolutions are faster than Winograd on modern CPUs, here is why
12. Lavin A, Gray S (2016) Fast algorithms for convolutional neural networks. In: 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp 4013–4021. Available: https://doi.org/10.1109/CVPR.2016.435
13. Lai L, Suda N, Chandra V (2018) Cmsis-nn: Efficient neural network kernels for arm cortex-m cpus. arXiv preprint arXiv:1801.06601
14. Sun D, Liu S, Gaudiot J-L (2017) Enabling embedded inference engine with arm compute library: A case study. arXiv preprint arXiv:1704.03751
15. Dukhan M Nnpack: an acceleration package for neural network computations. Available: https://github.com/Maratyszcza/NNPACK
16. Castelló A, Barrachina S, Dolz MF, Quintana-Ortí ES, Juan PS, Tomás AE (2022) High performance and energy efficient inference for deep learning on multicore arm processors using general optimization techniques and blis. Journal of Systems Architecture, vol 125, p 102459. Available: https://www.sciencedirect.com/science/article/pii/S1383762122000509
17. Bhattacharya S, Lane ND (2016) Sparsification and separation of deep learning layers for constrained resource inference on wearables. In: Proceedings of the 14th ACM Conference on Embedded Network Sensor Systems CD-ROM, pp 176–189
18. Lane ND, Bhattacharya S, Georgiev P, Forlivesi C, Jiao L, Qendro L, Kawsar F (2016) DeepX: A software accelerator for low-power deep learning inference on mobile devices. In: 2016 15th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN). IEEE, pp 1–12
19. Barrachina S, Castello A, Dolz MF, Low TM, Martinez H, Quintana-Orti ES, Sridhar U, Tomas AE Convdirect: A library with different implementations of the direct convolution operation. Available: https://github.com/hpca-uji/convDirect.git
20. Van Zee FG, van de Geijn RA (2015) BLIS: a framework for rapidly instantiating BLAS functionality. ACM Trans on Math Soft 41(3):14:1-14:33
21. Dolz MF, Barrachina S, Castello A, Quintana-Orti ES, Tomas AE Convwinograd: An implementation of the winograd-based convolution transform. Available: https://github.com/hpca-uji/convWinograd.git
22. Winograd S (1980) Arithmetic Complexity of Computations. Society for Industrial and Applied Mathematics
23. Barabasz B, Anderson A, Soodhalter KM, Gregg D (2020) Error analysis and improving the accuracy of Winograd convolution for deep neural networks. ACM Trans. Math. Softw. 46(4):1. Available: https://doi.org/10.1145/3412380
24. Dolz MF, Castelló A, Quintana-Ortí ES (2022) Towards portable realizations of Winograd-based convolution with vector intrinsics and OpenMP. In: 2022 30th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP), pp 39–46
25. Masliah I, Abdelfattah A, Haidar A, Tomov S, Falcou J, Dongarra J (2016) High-performance matrix-matrix multiplications of very small matrices. In: 22nd International European Conference on

Parallel and Distributed Computing (Euro-Par'16). Grenoble, France: Springer International Publishing, 2016-08
26. Tegra hardware information. Available: https://developer.nvidia.com/tegra-hardware-sales-inquiries
27. Jetson-stats is a package for monitoring and control the nvidia jetson. Available: https://github.com/rbonghi/jetson_stats
28. INA3221 Triple-Channel, High-Side Measurement, Shunt and Bus Voltage Monitor with I2C- and SMBUS-Compatible Interface, https://www.ti.com/product/INA3221#tech-docs
29. Barrachina S, Barreda M, Catalán S, Dolz MF, Fabregat G, Mayo R, Quintana-Ortí E (2013) An integrated framework for power-performance analysis of parallel scientific workloads. Energy, pp 114–119
30. Barrachina S, Castelló A, Catalán M, Dolz MF, Mestre JI (2021) A flexible research-oriented framework for distributed training of deep neural networks. In: 2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pp 730–739
31. Krizhevsky A, Sutskever I, Hinton GE (2012) 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., pp 1097–1105. Available: http://dl.acm.org/citation.cfm?id=2999134.2999257

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Authors and Affiliations

Manuel F. Dolz[1] · Sergio Barrachina[1] · Héctor Martínez[2] · Adrián Castelló[4] · Antonio Maciá[3] · Germán Fabregat[1] · Andrés E. Tomás[4]

Sergio Barrachina
barrachi@uji.es

Héctor Martínez
el2mapeh@uco.es

Adrián Castelló
adcastel@disca.upv.es

Antonio Maciá
a.macia@ua.es

Germán Fabregat
fabregat@uji.es

Andrés E. Tomás
antodo@upv.es

[1] Universitat Jaume I, Castelló de la Plana, Spain

[2] Universidad de Córdoba, Córdoba, Spain

[3] Universitat d'Alacant, Alacant, Spain

[4] Universitat Politècnica de València, València, Spain