Contents lists available at ScienceDirect

Journal of Systems Architecture

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

# High performance and energy efficient inference for deep learning on multicore ARM processors using general optimization techniques and BLIS

Adrián Castelló [a,*], Sergio Barrachina [b], Manuel F. Dolz [b], Enrique S. Quintana-Ortí [a], Pau San Juan [a], Andrés E. Tomás [b]

[a] Universitat Politècnica de València, Valencia, Spain
[b] Universitat Jaume I, Castellón de la Plana, Spain

ARTICLE INFO

*Keywords:*
Convolutional neural network
Inference
Multicore low-power processors

ABSTRACT

We evolve PyDTNN, a framework for distributed parallel training of Deep Neural Networks (DNNs), into an efficient inference tool for convolutional neural networks. Our optimization process on multicore ARM processors involves several high-level transformations of the original framework, such as the development and integration of Cython routines to exploit thread-level parallelism; the design and development of micro-kernels for the matrix multiplication, vectorized with ARM's NEON intrinsics, that can accommodate layer fusion; and the appropriate selection of several cache configuration parameters tailored to the memory hierarchy of the target ARM processors.

Our experiments evaluate both inference throughput (measured in processed images/s) and inference latency (i.e., time-to-response) as well as energy consumption per image when varying the level of thread parallelism and the processor power modes. The experiments with the new inference engine are reported for the ResNet50 v1.5 model on the ImageNet dataset from the MLPerf suite using the ARM v8.2 cores in the NVIDIA Jetson AGX Xavier board. These results show superior performance compared with the well-spread TFLite from Google and slightly inferior results when compared with ArmNN, the native library from ARM for DNN inference.

## 1. Introduction

Information technology companies are nowadays strongly interested in running deep learning (DL) models at the edge to improve security (safety and privacy), to accelerate the time-to-response (i.e., latency) experienced by the end-user, and to reduce the energy consumption for IoT (Internet-of-Things) applications [1–4]. The deployment of these trained DL models, known as *inference*, is often performed on a wide variety of user appliances, ranging from drones and mobile phones to wearables and IoT sensors [1–3]. In this scenarios, the inference process is computationally less expensive than the prior training stage, but often presents strict response time and/or energy constraints and has to be performed on devices with limited computational and memory capacities, as well as constraints in power supply.

In this paper, we investigate the efficient realization of an inference module for multicore ARM processors, extending our prior work in [5] to make the following contributions:

- Starting from the PyDTNN framework for distributed DNN training on clusters of computers [6], we obtain a basic module for inference based on the forward pass stage of the original framework, enhanced with some preliminary optimizations that replace a few key Python routines with efficient Cython-based counterparts parallelized with OpenMP directives.
- For the convolution layers, we follow our work in [7] to adopt a blocked variant of the IM2COL transform [8] that casts this operation into a matrix multiplication (GEMM) while eliminating the high memory costs of its conventional formulation. This is achieved via careful utilization of the packing routines in the BLIS [9] realization of the matrix multiplication kernel.
- We optimize the BLIS GEMM kernel with a dynamic mechanism to utilize architecture-specific cache configuration parameters. In addition, we expand the realization of GEMM in BLIS with a variant that targets the convolution operators to the most appropriate level of the cache.
- We develop a new micro-kernel for the BLIS realization of GEMM using NEON vector intrinsics for ARM processors. This opens the door to fuse (i.e., merge) consecutive layers, reducing the overhead of memory access for some DNN models.

* Corresponding author.
*E-mail address:* adcastel@disca.upv.es (A. Castelló).
[1] https://mlperf.org/inference-overview.

https://doi.org/10.1016/j.sysarc.2022.102459
Received 18 May 2021; Received in revised form 20 January 2022; Accepted 5 March 2022
Available online 15 March 2022

- We conduct a complete, incremental evaluation of the impact of each one of these contributions, using a standard use case from the MLPerf benchmark suite (v1.0)[1] for inference, on the ARM v8.2 Carmel processor embedded in the NVIDIA Jetson AGX Xavier board.
- Finally, we evaluate the performance of an alternative parallel scheme that prioritizes inference throughput versus latency. In addition, we complete the experimental analysis with a study of the energy consumption of these solutions, as this is a critical metric for many embedded devices.

In summary, our work demonstrates that using well-known, general high performance techniques one can attain a level of performance that is comparable to that of an architecture-specific solution as ArmNN, which exploits hardware details at very low level, or the state-of-the art inference module in Google TensorFlow Lite. We believe that sacrificing a small margin of performance in exchange for higher portability is a big contribution of our work. In this sense, our generic target is a "conventional" multicore processor. This decision is supported by the existence of a good number of devices in the market equipped with this type of processors but without GPU support.

The rest of the paper is structured as follows. In Section 2 we review and evaluate the prototype inference module obtained by applying a few basic optimizations to the code for the forward pass in PyDTNN. In Section 3 we describe several advanced optimizations introduced in this basic module. Next, in Section 4, we analyze other parallelization alternatives as well as energy consumption. Finally, in Section 5, we close the paper with a summarizing discussion and few concluding remarks.

## 2. Basic module for DL inference

In contrast with the training stage, the inference process is considerably less expensive though, when performed on edge devices, may present severe constraints in the response time, energy consumption, and/or memory requirements. In this section we describe our initial work to adapt the PyDTNN framework to common inference scenarios, targeting low power ARM-based architectures. To illustrate this process, we leverage the ResNet50 v1.5 model+ImageNet benchmark included in the MLPerf inference suite, focusing on the Carmel multicore processor (ARM v8.2) embedded in NVIDIA's Jetson AGX Xavier development kit.

### 2.1. The PyDTNN framework for DL

PyDTNN,[2] is a framework for distributed training of DNNs on clusters of computers, written in Python, that is designed as a research-oriented tool with a low learning curve. PyDTNN prioritizes simplicity to facilitate that users can adapt the framework to prototype research exploration; exposes an interface akin to that of popular DL packages such as Keras; and supports a significant part of common DNN models such as multi-layer perceptrons (MLPs), convolutional neural networks (CNNs), residual networks (ResNets), and transformers for natural language processing. In addition, PyDTNN offers validation accuracy and parallel performance, for DL training, on par with those attained by Google's TensorFlow [6]. To attain this, PyDTNN leverages high performance computational and communication libraries such as, for example, Intel's MKL; NVIDIA's cuDNN, cuBLAS and NCCL; as well as specialized implementations of MPI.

[2] The PyDTNN framework is available at https://github.com/hpca-uji/PyDTNN/ under a GNU General Public License v3.0.

### 2.2. ResNet50 v1.5 and ImageNet

CNNs are especially appropriate DL technologies for image recognition, recommendation systems, image classification, medical image analysis, natural language processing, brain–computer interfaces, financial time series, etc. The importance of CNNs is recognized by the MLPerf suite v1.0, which includes the ResNet50 v1.5 CNN model combined with the ImageNet ($224 \times 224$) dataset as one of its six benchmarks. This model consists of 176 layers, comprising a large number of four types of *transforms*: 53 convolutions, 48 non-linear (ReLU) functions, 52 batch normalizations, and 2 pooling operators; see [10] for details.

### 2.3. Jetson AGX Xavier

The NVIDIA Xavier board embeds an ARM Carmel 8-core CPU (with the ARMv8.2-FP16 extension), an NVIDIA 512-CUDA core Volta GPU, and 32 GiB of main memory. (As the target of this work is the optimization of inference on ARM architectures, we will not consider the GPU hereafter.) On the software side, the board runs under the Ubuntu Linux distribution 18.04.4, and includes the GNU C compiler gcc 10.0 and BLIS 0.7.0 for the linear algebra kernels.

For the experiments in this section, we employ all 8 ARM-based Carmel cores of the platform. Furthermore, we set the MAXN power mode available in the *nvpmodel* power/performance management utility included in the NVIDIA Jetson AGX Xavier. This mode activates all 8 Carmel cores and sets their frequency to 2.3 GHz. This permits an easier evaluation of the multi-threaded parallelization, as it avoids side effects that would occur if the system was allowed to automatically adjust the core frequencies depending on the workload and the number of active cores.

All tests in the paper employ IEEE 32-bit floating-point arithmetic (FP32). Furthermore, the tests were executed 20 times and the results were averaged to smooth out system load effects in the measurements. The variations observed between different executions were, in general, rather small.

### 2.4. Baseline inference

As a starting point for our work, we obtained a baseline module for inference (hereafter referred to as BASE) by simply utilizing the Python routine in PyDTNN for the training *forward pass* [11,12]. This prototype module presents the following features:

- The convolutions are cast in terms of GEMM kernels via an IM2COL re-organization of the activation inputs that constructs a large augmented matrix before invoking GEMM [8].
- The GEMM operation is realized via NumPy, which is linked against the ARM-optimized realization of this kernel in the BLIS framework [9]. This linear algebra library leverages multi-threaded parallelism using OpenMP and exploits the vector units in the ARM processor via an assembly-encoded micro-kernel with vector instructions. (BLIS is described in more detail in Section 3.)
- The batch normalizations are implemented as Python routines.
- The elementwise ReLU functions are encoded as Cython routines parallelized with OpenMP.
- The pooling layers are implemented as Cython routines that parallelize the IM2COL transform using OpenMP directives.

We first offer an analysis of the time costs of the inference process using the prototype inference module; see the column labeled as BASE in Table 1. Given the large number of transforms comprised by the ResNet50 v1.5 model (more than 150), we group the costs in the table into the four main types of transforms: 2D convolutions, batch normalizations, ReLU activation functions, and pooling operators. The costs reported there correspond to the seconds that are necessary to

**Table 1**
Cost analysis of the inference variants derived from PyDTNN when applied to ResNet50 v1.5+ImageNet and batch size $t = 128$ using the full 8 Carmel cores of NVIDIA's Jetson AGX Xavier. The individual cost is displayed only for the major components.

| Type of | Basic module (Section 2) | | GEMM optimizations (Section 3) | | |
|---|---|---|---|---|---|
| transform | BASE | CYTHON | CONV-opt | CACHE-opt | FUSE |
| Conv2D | 10.81 (44.82%) | 10.81 (81.34%) | 8.37 (77.14%) | 7.92 (78.49%) | 7.78 (87.21%) |
| Batch norm. | 10.58 (43.86%) | 0.55 (4.14%) | 0.55 (5.07%) | 0.55 (5.45%) | – |
| ReLU | 1.36 (5.64%) | 1.36 (10.23%) | 1.36 (12.53%) | 1.36 (13.47%) | 0.75 (8.41%) |
| Pooling | 1.08 (4.48%) | 0.26 (1.96%) | 0.26 (2.40%) | 0.26 (2.57%) | 0.26 (2.91%) |
| **Time (s)** | 24.08 | 13.63 | 10.85 | 10.09 | 8.92 |
| **Images/s** | 5.31 | 9.38 | 11.80 | 12.68 | 14.34 |

process a batch consisting of $t = 128$ images and the global throughput is also measured in number of images per second.

CNNs avoid overfitting by taking advantage of the hierarchical structure of the data. This is achieved via convolutional layers, which in general concentrate a significant fraction of the computational cost for CNNs. This expectation is confirmed only partially by the results in Table 1, which show that the convolutions consume 44.82% of the execution time of the initial BASE module. However, the Python realization of the batch normalizations is almost as expensive (43.86%). From this analysis, it is clear that, for this particular testbed (ResNet50 v1.5 with ImageNet and NVIDIA's Carmel processor), we should consider the convolutions and batch normalizations as the two first targets in the optimization of the baseline inference module.

### 2.5. Batch normalization

A primary optimization target corresponds to the batch normalizations, which, surprisingly, were responsible for almost 44% of the execution time in the BASE variant. An inspection of the Python code for the realization of this transform in PyDTNN, together with some additional experiments, guided us to conduct the following optimizations:

- Elimination of code invariants in this type of transforms, in particular, the calculation of the standard deviation and mean, which can be directly obtained from the prior training process. (This unnecessary recalculation was due to this routine being used for the training forward pass in PyDTNN, where these two parameters must be computed for each new batch.)
- Replacement of the Python code for this calculation with a Cython routine that is parallelized (via OpenMP) and vectorized (using the appropriate compiler directives).
- Avoidance of unnecessary accesses to large data arrays thanks to a more careful use of temporary variables.
- Adoption of column-major storage for the data arrays that conform to the accesses to these structures to favor a more efficient, vectorized data retrieval.

Fig. 1 illustrates the considerable benefits attained by the optimized realization versus the original one (respectively labeled as PYTH and CYTH in the figure). These results motivated us to apply a similar reformulation of the Cython-based pooling transforms in the ResNet50 v1.5 model, which were responsible for 4.48% of the total cost in the BASE module. (There exists a second pooling transform in the final layers model, but its cost is negligible.)

The positive impact of these optimizations is confirmed in the column labeled as CYTHON in Table 1, which shows that the time spent for the batch normalizations and pooling transforms is reduced from 10.58 s and 1.08 s in the BASE module to 0.55 s and 0.26 s, respectively, in the CYTHON module. As a result, in the new CYTHON variant, the batch normalization and pooling transforms contribute with much lower costs to the total execution time: 4.14% and 1.96%, respectively. The global outcome of this optimization is an acceleration of the processing throughput from 5.31 images/s for the former to 9.38 images/s for the latter (a speed-up of 1.76).

## 3. Optimization of GEMM-based convolutions for inference

After the initial optimization proposed in the CYTHON variant, the previous section identified the convolution as the key contributor to the practical cost of the inference module when applied to ResNet50 v1.5 in the target Jetson AGX Xavier platform. In this section, we describe some ARM architecture-aware optimization techniques that significantly reduce the cost of this operator. Prior to this, we open the section with a short review of the BLIS approach to obtain a portable, high-performance realization of GEMM.

### 3.1. BLIS: Open and portable kernels for dense linear algebra

Consider the GEMM operation $C \mathrel{+}= A \cdot B$, where $C \rightarrow m \times n$, $A \rightarrow m \times k$, and $B \rightarrow k \times n$. BLIS mimics GotoBLAS [13] to implement this kernel as three nested loops around a *macro-kernel* plus two *packing routines*. From that point, BLIS differs from GotoBLAS by decomposing the macro-kernel to expose two more loops around a micro-kernel; see Fig. 2 and [9] for details.

The architecture-specific optimization of the BLIS kernel requires a selection of the loop strides $m_c, n_c, k_c$ that matches the cache organization of the target processor [14] (plus the development of a vectorized version of the micro-kernel, to be discussed later). In some detail, the loop ordering in the BLIS realization of GEMM, together with the packing routines, and a proper selection of the cache configuration parameters/loop strides ($n_c$, $k_c$, $m_c$), orchestrate a regular pattern of data transfers, as illustrated in Fig. 3. Concretely, packing $B$ into $B_c$ inside loop L2 of the BLIS kernel in Fig. 2 makes a copy of these data into the L3 cache, and the re-use of this particular buffer for all iterations of the subsequent loop, L3, favors that this buffer persists in that level of the cache. Similarly, packing $A$ into $A_c$ in Loop L3 copies these data into the L2 cache and the repeated access to this buffer in loop L4 preserves it in that level. For a detailed discussion, see [9,14].

The development of an NVIDIA Carmel-specific micro-kernel using assembly code was done as part of the work in [15]. For a multi-threaded execution of the BLIS realization, the loops of the GEMM kernel to be parallelized can be selected at execution time. The OpenMP-based parallelization of the BLIS GEMM kernel has been previously analyzed for conventional multicore processors [16], modern many-threaded architectures [17], and low-power (asymmetric) ARM-based processors in [15].

### 3.2. Convolution via CONVGEMM

One major goal of packing in BLIS is to arrange the entries of $A$ and $B$ into $A_c$ and $B_c$, respectively, so that the elements of these buffers are accessed with unit stride when executing the micro-kernel [9]. In practice, provided $m$ is large enough, the cost of the packing for $B_c$ is negligible compared with the number of flops performed inside Loop L3. (A similar reasoning applies to the overhead due to the packing for $A_c$ and the value of $n_c$.)

In [7] we integrated the convolution within the BLIS [9] realization of GEMM, obtaining a CONVGEMM routine that reduces the considerable memory requirements of the full IM2COL transform. To attain this, the CONVGEMM realization assembles the augmented activation matrix *by blocks* with the novelty that, for performance reasons, the block dimensions are adjusted to the internal buffers utilized by BLIS GEMM to avoid the usage of extra memory while efficiently accommodating the data in the processor cache hierarchy; see [7] for the full details.

Fig. 4 displays the impact on performance when using the CONVGEMM routine versus the full-IM2COL approach for the convolutions appearing

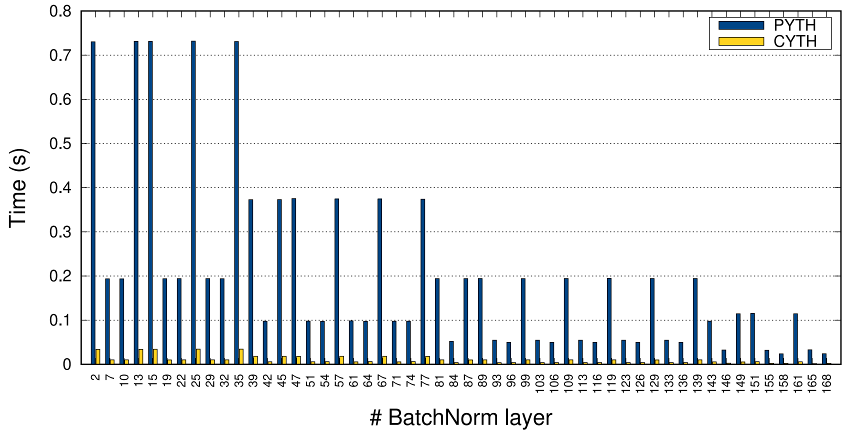

**Fig. 1.** Performance of the two options for batch normalization when applied to ResNet50 v1.5+ImageNet and batch size $t = 128$ using the full 8 Carmel cores of NVIDIA's Jetson AGX Xavier.

```
L1: for $j_c = 0, \ldots, n-1$ in steps of $n_c$
L2:     for $p_c = 0, \ldots, k-1$ in steps of $k_c$
            $B(p_c : p_c + k_c - 1, j_c : j_c + n_c - 1) \rightarrow B_c$          // Pack into $B_c$
L3:         for $i_c = 0, \ldots, m-1$ in steps of $m_c$
                $A(i_c : i_c + m_c - 1, p_c : p_c + k_c - 1) \rightarrow A_c$      // Pack into $A_c$
L4:             for $j_r = 0, \ldots, n_c - 1$ in steps of $n_r$                  // Macro-kernel
L5:                 for $i_r = 0, \ldots, m_c - 1$ in steps of $m_r$
                        $C_c(i_r : i_r + m_r - 1, j_r : j_r + n_r - 1)$      // Micro-kernel
                        $\mathrel{+}= A_c(i_r : i_r + m_r - 1, 0 : k_c - 1)$
                        $\cdot B_c(0 : k_c - 1, j_r : j_r + n_r - 1)$
```

**Fig. 2.** High performance implementation of GEMM in BLIS. $C_c \equiv C(i_c : i_c + m_c - 1, j_c : j_c + n_c - 1)$ is a notation artifact, introduced to ease the presentation of the algorithm. In contrast, $A_c, B_c$ denote buffers that are involved in data copies. For simplicity, we consider that $m, n, k$ are integer multiples of $m_c, n_c, k_c$ respectively, and $m_c, n_c$ are integer multiples of $m_r, n_r$ respectively.

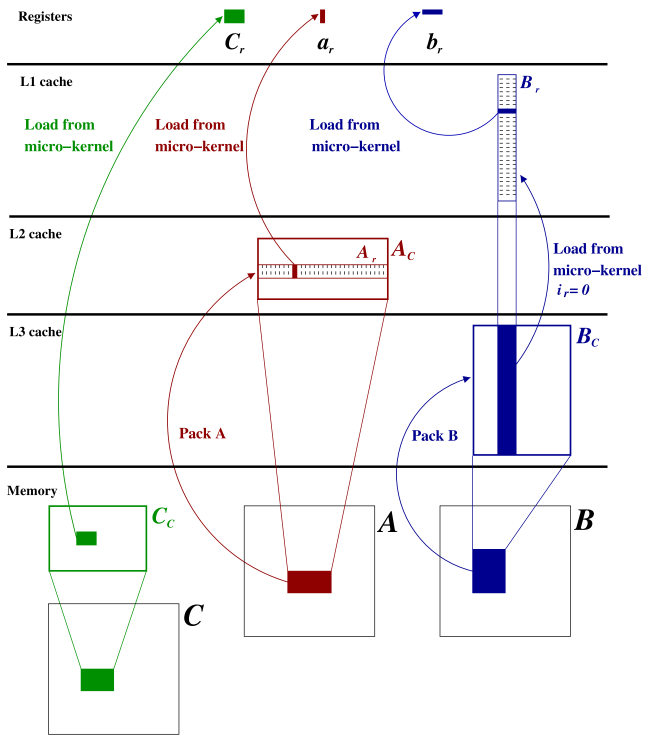

**Fig. 3.** Data movement in the BLIS implementation of GEMM.

in the testbed that is targeted in this section. Performance is measured there in terms of billions of floating-point operations, abbreviated as flops, per second (GFLOPS), taking into account the dimensions of the GEMM that is computed at each layer. For the two realizations of the convolution evaluated in the figure, IM2COL+GEMM employs the full IM2COL transform followed by an invocation to GEMM whereas CONVGEMM directly calls the CONVGEMM routine that integrates the memory-saving blockwise variant of IM2COL. In addition, for each layer, we include the GFLOPS attained by a direct invocation to GEMM (with operands of the same dimensions) that does not perform any type of IM2COL transform. This last rate offers an estimation of the overhead present in the realizations based on the full- and block-wise IM2COL. In an independent experiment, we could confirm that, in the latter case, this overhead is mostly due to some data re-organizations which are necessary to call CONVGEMM, not to the block-wise IM2COL itself.

The results in Fig. 4 expose that the best realization largely depends on the layer specifications (number of channels, number of filters, filter size, strides, etc.). The largest speed-up is achieved for layer 12 (1.68) with IM2COL+GEMM, while the smallest one is observed for layer 9 (1.02) when CONVGEMM is selected.

In general, those transforms for which both the kernel size and the number of channels are small (e.g., $1 \times 1$ kernels and 128 output channels), tend to favor IM2COL+GEMM while in the remaining cases CONVGEMM should be preferred. To accommodate this, in variant CONV-opt we fix the realization to utilize the most efficient option on a per-transform (layer) basis.

The column labeled as CONV-opt in Table 1 reports the performance of this second variant, which employs the best realization (IM2COL+GEMM or direct CONVGEMM) for each convolution transform of ResNet50 v1.5, showing a considerable increase in the processing throughput with respect to the CYTHON variant, from 9.63 to 11.80 images/s. This represents a speed-up of 1.22.

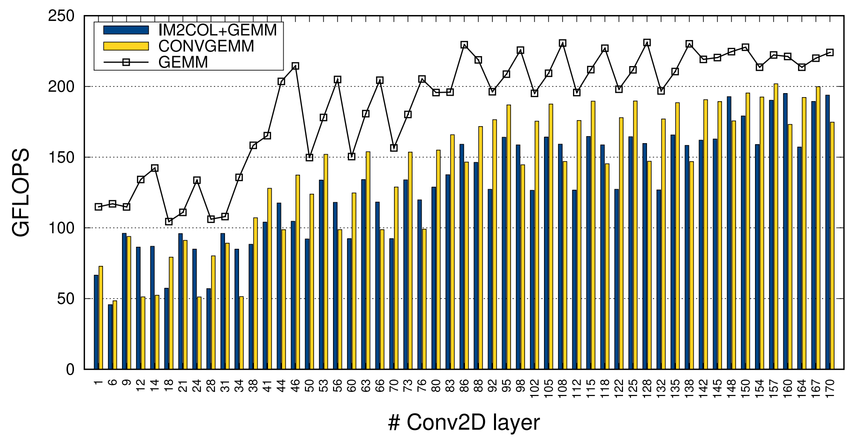

**Fig. 4.** Performance of the two options for the convolution when applied to ResNet50 v1.5+ImageNet and batch size $t = 128$ using the full 8 Carmel cores of NVIDIA's Jetson AGX Xavier.

### 3.3. Optimization of cache usage for GEMM

For performance and portability reasons, the values for the cache configuration parameters $n_c, k_c, m_c$ should be adjusted to the target processor cache hierarchy. This is done, in general, as part of an offline optimization process that prioritizes performance for moderate to large, "squarish" problems, with $m = n = k \in O(10^3)$. For the NVIDIA Carmel processor and FP32 arithmetic, this experimental optimization process in BLIS yields a selection of $m_c = 560, n_c = 3{,}072, k_c = 368$. These values ensure that $A_c$ fits into the L2 cache, and a panel of $B_c$ fits into the L1 cache. Furthermore, taking into account the associativity of these two cache levels, this selection aims to reduce the risk of evicting them from the corresponding level during the access to other data [14]. (At this point, we recall that the NVIDIA Carmel processor does not have an L3 cache.)

Unfortunately, many of the GEMM operations that are associated with the convolutions appearing in practical CNNs are far from presenting such "ideal" dimensions. Concretely, when tackled via the IM2COL transform, most of the initial convolutions in the target ResNet50 v1.5 model involve a matrix multiplication where $m, k$ are small (and equal), usually in $\{64, 128, 256\}$, while $n$ is much larger, of $O(10^4 - 10^5)$. The result is a suboptimal utilization of the cache memories and, in consequence, low performance. For example, for the matrix multiplication in the first convolution of ResNet50 v1.5, $m = k = 64$ while $n \approx 140K$. Therefore, $A_c$ is a small $64 \times 64$ matrix, which occupies only a minor fraction of the target 2-MiB L2 cache of the NVIDIA Carmel processor: Indeed, less than 1%!

To tackle this problem, we implemented our realization of the GEMM kernel, which follows the BLIS cache optimization principles, but allows a dynamic selection of $n_c, m_c, k_c$, at execution time, depending on the dimensions of the matrix multiplication appearing in each particular layer. In addition, we implemented a variant of the BLIS kernel that "swaps" the target cache levels for $A_c$ and $B_c$ (see Fig. 3), by interchanging loop L1 with L3, and loop L4 with L5 of the BLIS kernel. This favors that a panel of $A_c$ resides in the L1 cache while $B_c$ lies in the L2 cache. The purpose of this variant is to maximize the benefits of accessing data in the L2 cache. Finally, we performed an extensive experimental analysis to select the optimal values of these cache configuration parameters for the NVIDIA Carmel processor; and fixed the PyDTNN-based inference module to utilize the most efficient option on a per-transform (layer) basis.

Fig. 5 reports the performance, in GFLOPS, of the original version of BLIS, labeled as BLIS-BASE there, compared with those of the two new variants that dynamically adjust the cache parameters taking into account the parameters of each layer, with $A_c$ in the L2 cache and (part of) $B_c$ in the L1 cache (as in the original version of BLIS) or vice-versa. We identify these two variants in the figure as A2B1 and B2A1, respectively. In general, we observe that the best variant is highly dependent on the layer characteristics, with A2B1 offering a better option in more cases. The results also demonstrate that a careful selection of the variant outperforms the BLIS default option (BLIS-BASE) by a visible margin. The largest gain is observed for the second layer, where variant B2A1 yields a speed-up of 1.69 over BLIS-BASE, while the smallest gain appears in the penultimate layer, with a speed-up of less than 1.02. On average, the speed-up is 1.18 (arithmetic mean) and the weighted average speed-up is 1.21.

The global effect of the cache optimization techniques on the inference module is reported in the column labeled as CACHE-opt of Table 1. The results for the grouped Conv2D layers show a reduction of execution time from 8.37 s for CONV-opt to 7.92 s for CACHE-opt. This is lower than we could have estimated from the gains for GEMM due to the cache optimizations. To explain this, we note that the acceleration reported in Fig. 5 was observed for a standalone execution of the matrix multiplication kernel, that is, without the data reorganization that is necessary for the preparation of the IM2COL transform. In contrast, when these complete data transforms are taken into account, the overall gain due to this optimization is smaller, yielding a speed-up of 1.07, and an increase in the throughput rate from 11.80 to 12.68 images/s.

### 3.4. High-level micro-kernels for the NVIDIA Carmel processor

Let us now turn our attention to the micro-kernel. As shown in Fig. 2, this operation is responsible for performing the tiny matrix multiplication $C_r \mathrel{+}= A_r \cdot B_r$, where $C_r = C_c(i_r : i_r + m_r - 1, j_r : j_r + n_r - 1) \rightarrow m_r \times n_r$; $A_r = A_c(i_r : i_r + m_r - 1, 0 : k_c - 1) \rightarrow m_r \times k_c$; and $B_r = B_c(0 : k_c - 1, j_r : j_r + n_r - 1) \rightarrow k_c \times n_r$. In practice, the micro-kernel is implemented as a simple loop along the $k_c$ dimension of the product that updates $C_r$ with the outer product of one column of $A_r$ and one row of $B_r$ per iteration:

```
for $k = 0, \ldots, k_c - 1$ in steps of $1$          Micro-kernel
    $C_r \mathrel{+}= A_r(:, k) \cdot B_r(k, :)$
```

All routines of the BLIS-inspired realization of GEMM are encoded in plain C except for the micro-kernel. This enhances portability as migrating the kernel to a particular processor architecture only needs to develop an efficient realization of that component for the target processor. The micro-kernel is usually vectorized using either architecture-dependent assembly instructions or vector intrinsics [9]. As a rule of thumb, $m_r, n_r$ are selected so that $C_r$, a column of $A_r$, and a row of $B_r$ occupy a significant fraction of the processor (vector) registers. Besides, $k_c$ is set so that the cost of loading $C_r$ into the processor registers (before the loop commences) and writing back the updated block into the main memory (once the loop is completed) is amortized over a

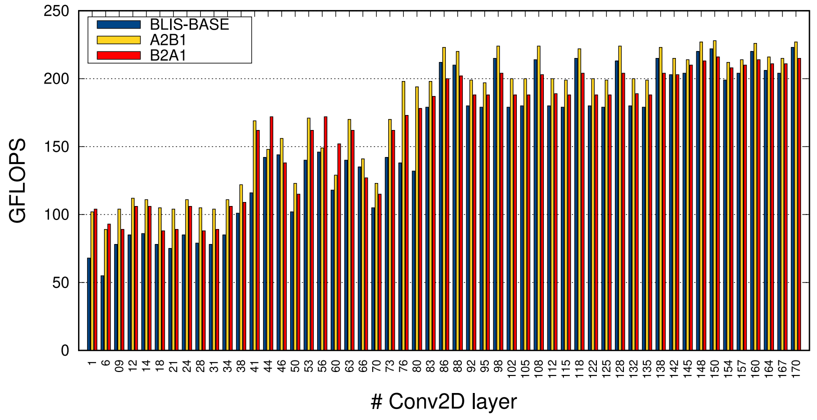

**Fig. 5.** Performance of the matrix multiplications appearing in the convolution layers of ResNet50 v1.5+ImageNet with batch size $t = 128$ using the full 8 Carmel cores of NVIDIA's Jetson AGX Xavier.

sufficient amount of flops; see [14]. These conditions usually imply that $m_r, n_r \in O(10)$ and $k_c \in O(100)$.

For the ARM-based processors targeted in our work, we developed several implementations of the micro-kernel using ARM NEON intrinsics. The best results were in general obtained for a micro-kernel with $m_r \times n_r = 8 \times 8$, which loads the full $C_r$ into 16 vector registers (with 4 FP32 elements per 128-bit vector register). In addition, the implementation integrates data pre-fetching [18] by loading the first column/row of $A_r/B_r$ into 2 + 2 vector registers before the micro-kernel loop commences. Then, at each iteration of the loop, say $k$, the micro-kernel prefetches the $(k+1)$-th column/row of $A_r/B_r$ into 2 + 2 additional vector registers, to then proceed to accumulate in $C_r$ the product involving the $k$th column/row using vector instructions.

Our NEON-based implementation of the micro-kernel compiled with GNU `gcc-10` and the appropriate optimization flags delivered a sustained performance that is similar to that of the original assembly-encoded micro-kernel in BLIS for ARM architectures.

The key observation though is that a "high-level" implementation of the micro-kernel, using plain C code plus ARM NEON intrinsics, paves the road to an efficient fusion of different types of layers, as described in the following subsection.

### *3.5. Layer fusion*

The ReLU function is a simple test-and-set operator that is applied element-wise to the activations of a layer, setting to zero all negative values and leaving the remaining ones unchanged. The batch normalization involves a couple of shift and scale arithmetic operations involving mean and standard deviation parameters. From the computational point of view, the arithmetic cost of these two types of transforms is low compared with the convolution itself. Therefore, their contribution to the total time in Table 1 seems excessive, especially when considering the Cythonized, parallelized, and vectorized version of the kernels.

Some additional experiments allowed us to identify that these costs are mostly due to memory accesses. To tackle this, when possible, we fuse (i.e., merge) the application of the batch normalization and ReLU with a previous convolution into a single "multi-layer operation". For this purpose, we take advantage of the high-level implementation of the micro-kernel (using C code enhanced with ARM NEON intrinsics) to integrate the ReLU function and batch normalization as part of the micro-kernel code. For the ReLU function, this is straightforward as this transform is applied element-wise. For the batch normalization, the fusion is more involved as the 2D output of the matrix multiplication needs to be mapped into the 4D result of the convolution as part of the fused normalization.

From the implementation point of view, the specialized case of the micro-kernel with fused layers is invoked from GEMM to update matrix $C$ during the final iteration of the loop that traverses the $k$-dimension of the problem (indexed by $p_c$); see Fig. 2. For the remaining iterations of that loop, the GEMM realization invokes the regular (i.e., non-fused) micro-kernel.

The FUSE column in Table 1 shows a cost for the fused Conv2D operations of 7.78 s only, which is even lower than that of the non-fused operations in CACHE-opt. This is due to the use of the new micro-kernel, with vector intrinsics. This reduction is even more notable if we consider that this micro-kernel not only performs the Conv2D operations but also all the batch normalizations as well as the ReLU functions which could be fused (about half of them). As a result, we obtain a raise in the throughput rate from 12.68 s for CACHE-opt to 14.34 s for FUSE, which corresponds to a speed-up of 1.13.

## 4. General evaluation

### *4.1. Comparison with other frameworks*

The global result of the multi-step optimization process described in the previous sections shows an increase in the processing rate from the original 5.31 images/s with the prototype inference module derived from PyDTNN up to 14.34 images/s, which yields a global speed-up of 2.70 over the original solution.

Table 2 compares the throughput of the optimized inference module based on PyDTNN with the state-of-the-art results reported in the latest release of the MLPerf benchmark (1.0) using the Carmel processor in the NVIDIA's Jetson AGX Xavier platform.[3] Overall, (one instance of) PyDTNN outperforms TFLite while being slower than the native ArmNN. At this point though, we would like to remark that the optimization of PyDTNN was achieved via high-level transformations of the original Python code for PyDTNN, the development of some high-level Cython routines, the encoding of a NEON-based micro-kernel with fused layers, and the appropriate selection of some configuration parameters. Therefore, we believe that our techniques are quite general, yielding a moderately portable inference engine to different platforms from NVIDIA/ARM as well as from other vendors.

### *4.2. Real-time inference and alternative parallelization schemes*

In time-constrained scenarios, raw inference throughput (i.e., processed images/s) is a secondary figure-of-merit and the evaluation should be instead focused on the time-to-response. The plot in the top-left of Fig. 6 reports the time to process a batch consisting of a varying number of images, from $t = 1$ to 128, using the full eight ARM cores

[3] https://mlperf.org/inference-results/.

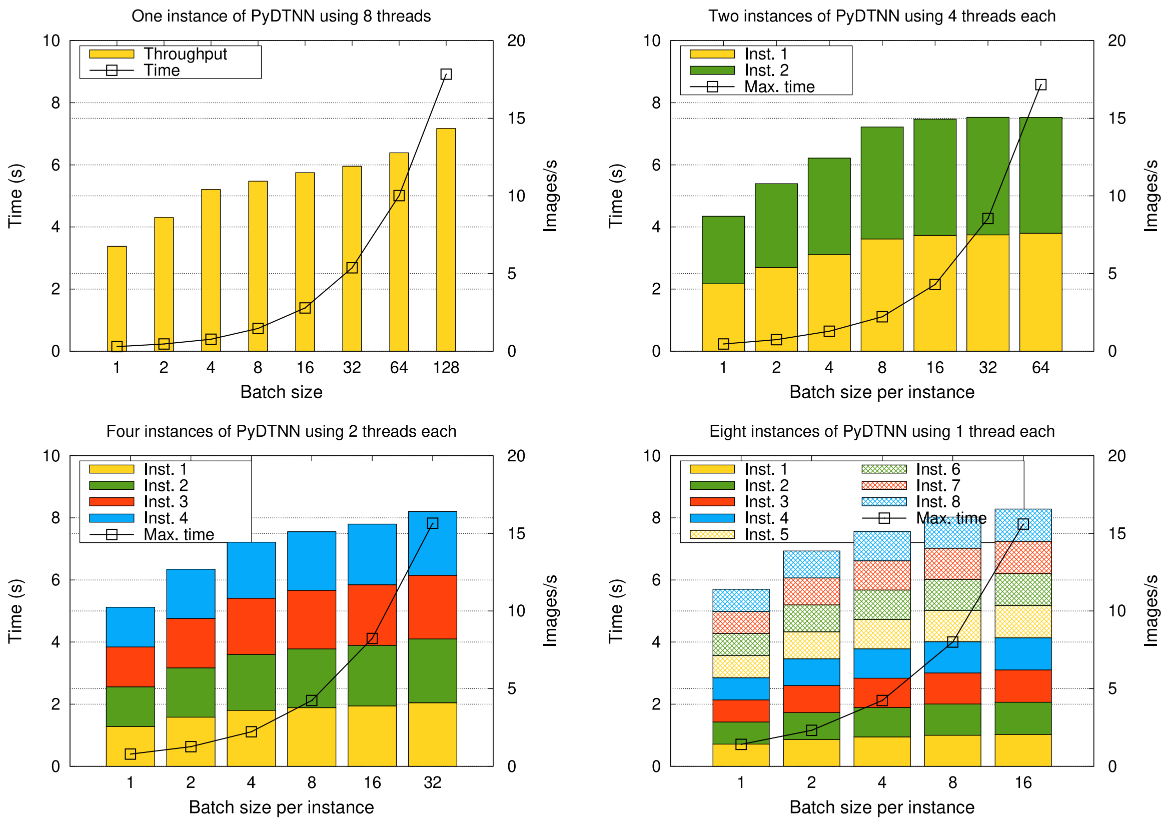

**Fig. 6.** Maximum inference time (for all instances) and throughput (in images/s) when running 1, 2, 4, and 8 concurrent instances of PyDTNN respectively using 8, 4, 2, and 1 threads each (top-left, top-right, bottom-left, and bottom-right, respectively).

**Table 2**
Throughput comparison of inference frameworks using the full 8 Carmel cores of NVIDIA's Jetson AGX Xavier.

| Framework | Version | Images/s | MLPerf | Batch size (images) |
|---|---|---|---|---|
| TFLite | 2.4.1 (ruy) | 13 | 1.0 | Unknown |
| ArmNN | 21.02 (Neon) | 18 | 1.0 | Unknown |
| PyDTNN (1 instance) | 1.0 | 14.34 | – | 128 |
| PyDTNN (4 instances) | 1.0 | 16.40 | – | 32/instance |
| PyDTNN (8 instances) | 1.0 | 16.56 | – | 16/instance |

in the NVIDIA Jetson board. This experiment shows that increasing the batch size gradually improves the throughput (images/s) of the inference engine, especially for small values of $t$, but also augments the total execution time for the batch. Thus, depending on the specific constraint determined by a given application on the time-to-response, the results identify the maximum batch size that can be used while still meeting that threshold and maximizing productivity. For instance, an upper bound of 2 s in the time-to-response is met for a batch with $t = 16$, but not for $t = 32$.

Related to the time-to-response versus throughput balance, there is an alternative parallelization scheme that is worth investigating: concretely, instead of executing a single instance of PyDTNN that utilizes all 8 platform cores (via multi-threading), we can run multiple instances of the inference engine, each mapped to a distinct subset of the platform cores and working with a separate batch of images (provided the memory capacity allows to replicate the model). The rest of the plots in Fig. 6 show the results for this experiment using 2, 4, and 8 instances of PyDTNN, which respectively utilize 4, 2, and 1 distinct core(s) each, thus involving the full 8 ARM cores of the platform. This experiment offers a few interesting insights:

1. The global throughput of this alternative scheme offers higher performance than the option that runs a single instance of PyDTNN. Specifically, compared with the 14.34 images/s of the conventional parallelization approach, running 2 concurrent instances of PyDTNN delivers 15 images/s, 4 concurrent instances achieves 16.4 images/s, and scaling up to 8 instances provides 16.56 images/s, standing rather close to the 18 images/s delivered by ArmNN. In consequence, provided a large number of images are available at an "input inference queue", we may want to process them using multiple instances of the inference engine. Also, this parallelization scheme is particularly interesting in ensemble learning, where the predictions from multiple DNNs trained on different initial conditions (e.g. weights initialization) are combined to reduce the variance and generalization error of the predictions. Notice that in this scenario the same batch of images is simultaneously passed for inference to the different DNNs in the ensemble.
2. All instances mostly deliver the same throughput, which demonstrates that the overheads due to the concurrent execution of multiple concurrent instances are negligible. These are good news for multi-model scenarios as those arising in autonomous vehicles.
3. For real-time scenarios, increasing the number of concurrent instances of PyDTNN penalizes the time-to-response of a single batch. For example, a batch $B_1$, consisting of 16 images, is processed in 1.4 s when executed using a single instance of PyDTNN using the full board resources. In comparison, the same batch $B_1$ takes 2.14 s to process when there are two running instances of PyDTNN, one working on this batch and a second one processing a different batch $B_2$. When the number of instances of PyDTNN is raised to 4, the time is almost doubled, and it takes 4.11 s to process $B_1$. In the latter case, when 8 instances are concurrently launched, the processing of the batch

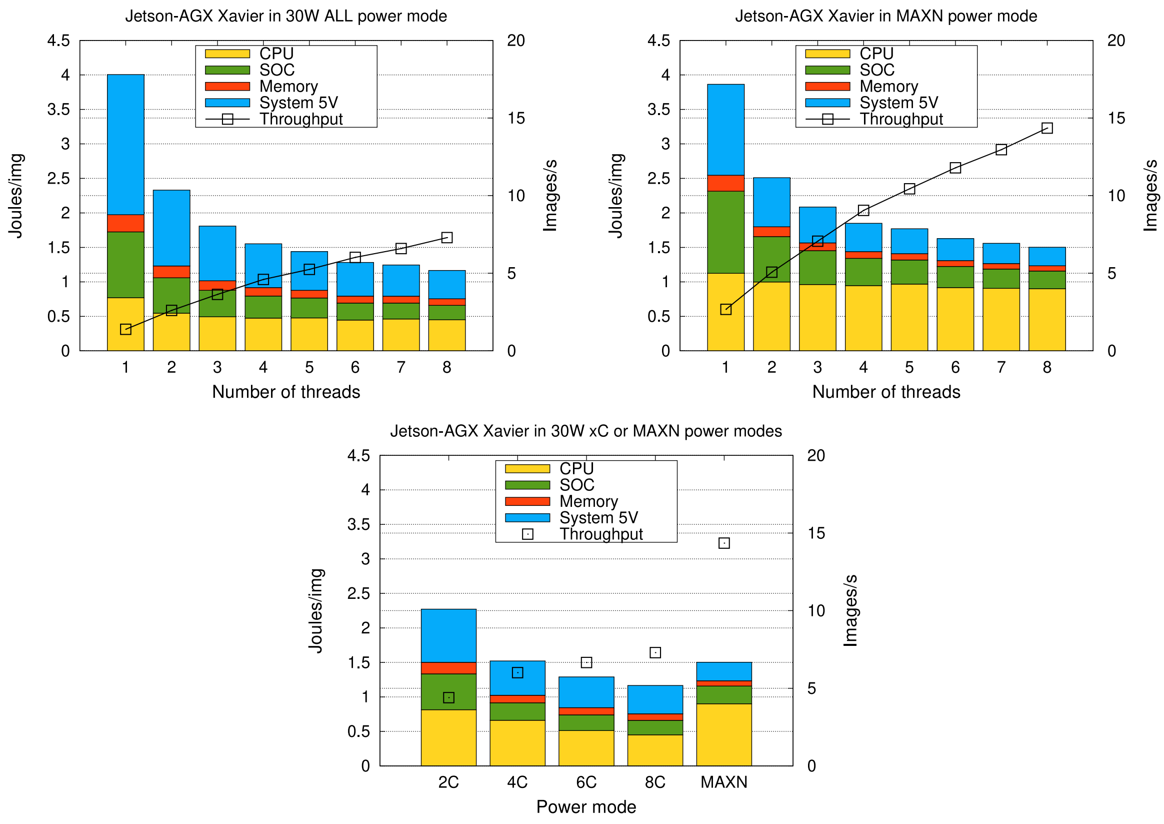

**Fig. 7.** Inference energy consumption per image and throughput (in images/s) when running 1 instance of PyDTNN with the NVIDIA Jetson AGX Xavier in two power modes, 30 W ALL and MAXN (top-left and top-right, respectively) using 1–8 cores; and using the 30 W × C power modes which deactivate some of the socket cores (8 − x) versus the MAXN power mode.

takes 7.8 s. This was about to be expected as, in the latter case, the amount of resources dedicated to processing that particular batch is divided by 8, 4 or 2, compared with the single-instance, 2-instance, and 4-instance execution, respectively.

### *4.3. Energy consumption, power modes, and performance*

Energy consumption is a relevant metric for energy-constrained platforms, such as battery-powered embedded devices. In the following experiment, we evaluate the energy consumption per image attained with the most significant power modes in the Jetson AGX Xavier board: 30 W ALL and MAXN. In the former case, the operating system regulates the hardware elements in the board (in particular, the frequency of the CPU cores and the GPU) to ensure that the system does not exceed a threshold power dissipation of 30 watts. In practice, we observed that, for example, this results in the processor frequency for the ARM cores being set to 1.2 GHz when all 8 cores are utilized. In the latter power mode, the frequency is set to 2.3 GHz for all ARM cores.

The top two plots in Fig. 7 show the energy consumption per image when running a single instance of PyDTNN that processes a batch of $t = 128$ images using an increasing number of threads, from 1 to 8, with the board operating in the 30 W ALL or MAXN power modes. We can observe the significant impact of the power modes on the inference throughput, which is roughly multiplied by two for MAXN compared with 30 W ALL. This is natural if we take into account the increase of the CPU frequency which is also mostly doubled from 1.2 GHz to 2.3 GHz. However, the performance boost comes at the cost of some energy efficiency loss, as the plots show that, for example, 8 threads running in 30 W ALL consume about 1.2 joules/image while the same cores in MAXN increase energy consumption to 1.5 joules/image.

To close the analysis of energy consumption, the bottom plot in Fig. 7 reports the energy consumption per image and throughput when we employ the alternative 30 W xC power mode, where x specifies the number of active cores while the remaining 8 − x cores are switched off. Note that this is different from the 30 W ALL power mode as in this later configuration all cores are turned on, though some of them may be idle (C state) because they do not intervene in the execution of the inference module. Due to that, the CPU frequency of the active cores can be raised beyond 1.2 GHz up to a higher clock rate, without exceeding the power budget of 30 W. For instance, comparing the energy consumption (per image) of the 30 W 2C power mode with that observed for the execution using two cores in 30 W ALL power mode, we can appreciate a slightly higher consumption for the latter, which was to be expected as idle cores still consume some energy. Given the CPU frequency increase in 30 W 2C, the inference throughput of 2.4 images/s using 2 cores in the 30 W ALL mode is almost doubled to 4.5 images/s in 30 W 2C.

## 5. General discussion and concluding remarks

In this paper, we have evolved PyDTNN, a framework for distributed parallel training of Deep Neural Networks (DNNs), to efficiently perform inference with convolutional neural networks on multicore ARM processors. This new inference engine inherits the appealing features of the ancestor PyDTNN training framework in terms of simplicity, user-friendly interface, and support for popular DNNs such as MLPs, CNNs, ResNets, and transformers. In addition, this inference tool applies some general-purpose high performance techniques as well as a few architecture-specific optimizations to deliver inference throughput that is competitive with that observed for popular frameworks, such as TF Lite, as well as highly architecture-dependent counterparts for ARM-based processors such as ArmNN.

Our work incrementally introduces the following (mostly architecture-agnostic) optimizations to the baseline implementation of the inference module in PyDTNN:

1. Replace Python with Cython-accelerated code for certain compute-intensive routines.
2. Replace the conventional IM2COL+GEMM convolution with an alternative that embeds the IM2COL transform within the packing (re-organization) that is implicitly done in the BLIS realization of GEMM.
3. Adjust the cache configuration parameters for the BLIS realization of GEMM to better match the dimensions of the matrix multiplications arising in the Resnet50 v1.5 model.
4. Fuse convolution layers with subsequent ReLU transforms and batch normalizations.

Among these, the first optimization does not necessarily improve cache usage while the second, third and fourth ones improve performance by reducing the number of memory accesses and better utilizing the cache hierarchy [14].

We note that a major part of the optimization techniques described in this paper are portable to other computer architectures (and are applicable to other DNN models). BLIS itself is highly portable as well as it is composed of C code except for a small micro-kernel, which can be encoded directly in assembly or in C using vector intrinsics for high performance, and there exist high performance realizations of the micro-kernel for many of the current multicore processors.

Also, the results for the multi-instance parallelization scenario demonstrate the relative importance of the batch size in the time-to-response versus raw throughput (images/s). Also, the experiments evaluating the energy consumption in the different modes reveal that a higher throughput comes at the cost of higher energy consumption. However, disabling cores helps to increase the throughput at a constant power budget.

## 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.

## Acknowledgments

This research was partially sponsored by projects TIN2017-82972-R of *Ministerio de Ciencia, Innovación y Universidades*, Spain and Prometeo/2019/109 of the *Generalitat Valenciana*, Spain. Adrián Castelló was supported by the Juan de la Cierva-Formación project FJC2019-039222-I of the *Ministerio de Ciencia, Innovación y Universidades*, Spain. Manuel F. Dolz was also supported by the Plan GenT project CDEIGENT/2018/014 of the *Generalitat Valenciana*, Spain.

## References

[1] K. Hazelwood, et al., Applied machine learning at Facebook: A datacenter infrastructure perspective, in: 2018 IEEE Int. Symp. High Performance Computer Architecture (HPCA), 2018, pp. 620–629.
[2] J. Park, et al., Deep learning inference in Facebook data centers: Characterization, performance optimizations and hardware implications, 2018, arXiv:1811.09886.
[3] C. Wu, et al., Machine learning at facebook: Understanding inference at the edge, in: 2019 IEEE Int. Symp. High Performance Computer Architecture (HPCA), 2019, pp. 331–344.
[4] S. Yi, C. Li, Q. Li, A survey of fog computing: Concepts, applications and issues, in: Proceedings of the 2015 Workshop on Mobile Big Data, Mobidata '15, Association for Computing Machinery, New York, NY, USA, 2015, pp. 37–42.
[5] P.S. Juan, P. Alonso-Jordá, E.S. Quintana-Ortí, High performance and energy efficient integer matrix multiplication for deep learning, in: 2021 29th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), 2021, pp. 122–125, http://dx.doi.org/10.1109/PDP52278.2021.00027.
[6] S. Barrachina, A. Castelló, M. Catalán, M.F. Dolz, J.I. Mestre, PyDTNN: A user-friendly and extensible framework for distributed deep learning, J. Supercomput. (2021) http://dx.doi.org/10.1007/s11227-021-03673-z.
[7] 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: Proc. 32nd Int. Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), 2020, pp. 91–98.
[8] K. Chellapilla, S. Puri, P. Simard, High performance convolutional neural networks for document processing, in: G. Lorette (Ed.), Tenth International Workshop on Frontiers in Handwriting Recognition, Université de Rennes 1, Suvisoft, La Baule (France), 2006, http://www.suvisoft.com. URL https://hal.inria.fr/inria-00112631.
[9] 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.
[10] P. Goyal, P. Dollár, R. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, K. He, Accurate, large minibatch SGD: Training ImageNet in 1 hour, 2017, arXiv:1706.02677.
[11] V. Sze, Y.-H. Chen, T.-J. Yang, J. Emer, Efficient processing of deep neural networks: A tutorial and survey, Proc. IEEE 105 (12) (2017) 2295–2329.
[12] S. Pouyanfar, et al., A survey on deep learning: Algorithms, techniques, and applications, ACM Comput. Surv. 51 (5) (2018) 92:1–92:36.
[13] K. Goto, R. van de Geijn, Anatomy of high-performance matrix multiplication, ACM Trans. Math. Softw. 34 (3) (2008) 12:1–12:25.
[14] T.M. Low, F.D. Igual, T.M. Smith, E.S. Quintana-Orti, Analytical modeling is enough for high-performance BLIS, ACM Trans. Math. Softw. 43 (2) (2016) 12:1–12:18.
[15] S. Catalán, et al., Architecture-aware configuration and scheduling of matrix multiplication on asymmetric multicore processors, Cluster Comput. 19 (3) (2016) 1037–1051.
[16] F.G.V. Zee, et al., The BLIS framework: Experiments in portability, ACM Trans. Math. Softw. 42 (2) (2016).
[17] T.M. Smith, et al., Anatomy of high-performance many-threaded matrix multiplication, in: Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium, IPDPS '14, IEEE Computer Society, USA, 2014, pp. 1049–1059.
[18] J. Hennessy, D. Patterson, Computer Architecture: A Quantitative Approach, third ed., Morgan Kaufman, 2003.