Journal of Parallel and Distributed Computing 167 (2022) 240–254

Contents lists available at ScienceDirect

Journal of Parallel and Distributed Computing

www.elsevier.com/locate/jpdc

# Efficient and portable GEMM-based convolution operators for deep neural network training on multicore processors

Sergio Barrachina [a], Manuel F. Dolz [a,*], Pablo San Juan [b], Enrique S. Quintana-Ortí [c]

[a] Depto. de Ingeniería y Ciencia de Computadores, Universidad Jaume I, Castellón, Spain
[b] Depto. de Sistemas Informáticos y Computación, Universitat Politècnica de València, Spain
[c] Depto. de Informática de Sistemas y Computadores, Universitat Politècnica de València, Spain

ARTICLE INFO

*Article history:*
Received 27 February 2021
Received in revised form 4 April 2022
Accepted 21 May 2022
Available online 30 May 2022

*Keywords:*
Convolutional neural networks
Distributed training
High performance
Python
Clusters of multicore processors

ABSTRACT

Convolutional Neural Networks (CNNs) play a crucial role in many image recognition and classification tasks, recommender systems, brain-computer interfaces, etc. As a consequence, there is a notable interest in developing high performance realizations of the convolution operators, which concentrate a significant portion of the computational cost of this type of neural networks.

In a previous work, we introduced a portable, high performance convolution algorithm, based on the BLIS realization of matrix multiplication, which eliminates most of the runtime and memory overheads that impair the performance of the convolution operators appearing in the forward training pass, when performed via explicit IM2COL transform. In this paper, we extend our ideas to the full training process of CNNs on multicore processors, proposing new high performance strategies to tackle the convolution operators that are present in the more complex backward pass of the training process, while maintaining the portability of the realizations. In addition, we conduct a full integration of these algorithms into a framework for distributed training of CNNs on clusters of computers, providing a complete experimental evaluation of the actual benefits in terms of both performance and memory consumption. Compared with baseline implementation, the use of the new convolution operators using pre-allocated memory can accelerate the training by a factor of about 6%–25%, provided there is sufficient memory available. In comparison, the operator variants that do not rely on persistent memory can save up to 70% of memory.



## 1. Introduction

Machine learning via deep neural networks (DNNs) is being increasingly adopted to tackle a large variety of applications beyond the traditional cubbyhole of this technology in image recognition and signal processing [15,5,22,28,35]. To improve their accuracy, many of the DNN models designed for these applications need to be trained over an extremely large amount of data [24]. For the same reason, both the dimension (number of layers) and complexity of DNNs are growing rapidly, and recent models involve up to billions of parameters [13,27,6]. In consequence, DNN training is currently conducted using distributed algorithms on high performance computing (HPC) facilities, with dozens or even hundreds of nodes, sometimes enhanced with fast memory modules (high bandwidth memory, or HBM) and high performance interconnection networks (e.g., Infiniband) [22]. At this initial point, we recognize that DNN training can significantly be accelerated using hardware accelerators, such as AMD/NVIDIA's graphics processors (GPUs) or Google's tensor processing units (TPUs). However, we also point out the interest of companies like Facebook to exploit idle workload cycles in their HPC facilities, which leave a significant number of (general-purpose) multicore CPUs to perform distributed model (re-)training during off-peak periods [15]. In this paper, we address how to leverage efficiently these idle resources to perform distributed training of CNNs.

Convolutional (deep) neural networks (CNNs) are a specialized type of multilayer perceptrons with application in image recognition, recommender systems, image classification, medical image analysis, natural language processing, brain-computer interfaces, and financial time series, among others. CNNs exhibit an implicit regularization that takes advantage of the hierarchical structure of the data in order to avoid overfitting. This property is achieved via the application of *convolution operators*, which concentrate a significant fraction of the computational cost for CNNs.

A flexible, reliable, and, in many cases, high performance approach to realize a convolution operator consists in applying the IM2COL transform [10] to the layer activation inputs, followed by a general matrix multiplication (GEMM) that takes advantage of

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

https://doi.org/10.1016/j.jpdc.2022.05.009

optimized realizations of this computational kernel in high performance linear algebra libraries (e.g., Intel MKL, OpenBLAS, GotoBLAS2, BLIS, ATLAS, etc.). Unfortunately, there are two major problems with this approach: 1) a large memory workspace is required to host the intermediate matrix generated by the IM2COL transform; and, especially for training, 2) the time to apply this transform is not negligible for complex CNNs. In [26], we presented a portable high performance convolution algorithm based on the BLIS [33] realization of GEMM, named CONVGEMM, that practically eliminates the memory and time cost of the IM2COL transform, while maintaining the portability and performance of the underlying realization of the BLIS GEMM for multicore processors.

In this paper, we extend our previous work in [26] to obtain an efficient integration of the convolution operators in a framework for distributed training of DNNs on clusters of computers equipped with multicore processors. In particular, this work makes the following contributions:

- For the computation of the downstream gradients with respect to the inputs, we adopt an approach similar to that in [26] to integrate a COL2IM operator within the BLIS realization of GEMM, yielding a DECONVGEMM operator that avoids the creation of a large intermediate matrix, while maintaining the portability and, to a certain extent, performance.
- For the computation of the downstream gradients with respect to the filters, we leverage the existing CONVGEMM algorithm via a novel REINDEX transform that re-arranges one of the input matrices to avoid the explicit operation with the transposed of that operand. As an alternative, we also address this calculation via a transposed variant of the CONVGEMM operator.
- We complete the general discussion of the CONVGEMM and DECONVGEMM operators and associated transforms with a detailed description using high level code that should allow to reproduce our implementations. In addition, we discuss several variations of the convolution operators that trade off memory consumption for performance.
- We integrate the resulting CONVGEMM/DECONVGEMM algorithms and the REINDEX transform into PyDTNN (*Python Distributed Training of Neural Networks*), a deep learning framework that offers a fair combination of functionality, computational performance, and friendly interface, prioritizing flexibility to prototype new research ideas.
- We perform a complete experimental evaluation showing the performance advantages and memory savings over the baseline approach that operates with large workspaces using the explicit IM2COL and COL2IM transforms. This evaluation includes four popular DNNs (ResNet34, VGG16, DenseNet121, and GoogLeNet) and two datasets (CIFAR-10 and ImageNet), and targets a state-of-the-art cluster equipped with Intel Xeon Gold 5120 processors with the nodes connected via an Infiniband EDR network.

The rest of the paper is structured as follows. In Section 2, we offer a short review of supervised iterative training for DNNs via the stochastic gradient descent (SGD), paying special attention to the convolution layers appearing in the forward-backward iteration. In Section 3, we first discuss the BLIS kernel for GEMM and explain how this was leveraged in [26] to obtain an efficient realization of the convolution operator in the forward pass of the training process. In the same section, we next discuss how to extend that idea to the more complex backward propagation stage, which constitutes one of the major contributions of this work. In Section 4, we elaborate on the integration of the new operators and transforms into PyDTNN, and assess the benefits of the result via a complete experimental validation. In Section 5, we revisit some related work on GEMM-based operators and compare them with those presented in this work. Finally, in Section 6, we close the paper with a short summary and a collection of concluding remarks.

## 2. Convolutional neural networks

### 2.1. Overview of supervised training

Consider a collection of *input vectors*, denoted by $x_1, x_2, \ldots, x_s \in \mathbb{R}^m$, with associated *output vectors* $y_1, y_2, \ldots, y_s \in \mathbb{R}^n$ (also known as *labels* or ground truth). From the mathematical perspective, a DNN defines a non-linear function $\mathcal{F}: \mathbb{R}^m \rightarrow \mathbb{R}^n$ so that $\mathcal{F}(x_r) = \tilde{y}_r$, where we expect that $\tilde{y}_r \approx y_r$, for $r = 1, 2, \ldots, s$; see [30,18,23].

In supervised training, the DNN "learns" from the labeled data, adjusting its model parameters (that is, weights and biases) to diminish the difference (deviation or error) between the ground truth and the output computed by the DNN as part of the forward pass (FP); that is, reduces the "error" $\|y_r - \tilde{y}_r\|$ (for all $r$). This learning process is usually conducted via the SGD method (or some related variant) [18], which is an iterative "back-propagation" (BP) procedure that, given an output-label pair, computes the gradients for the DNN parameters that minimize this difference. The gradients are then used to update the DNN parameters, in preparation for the next FP-BP iteration, with a new input sample; see, e.g. [18]. These two stages of BP are referred to in our work as gradient computation (BP-GC) and weight updates (BP-WU).

The convergence of the training process is accelerated in practice by applying the FP-BP iteration in batches of $t$ samples at a time, for a moderate value $t$ in the order of few hundreds. This also helps to overcome the memory bandwidth constraints of current computer architectures, by turning training into a compute-bound process. On the negative side, augmenting the batch size may affect the convergence and accuracy of the training process, often requiring a complex and application-dependent tuning of the learning rate, which needs to be dynamically adjusted as the training process evolves [36].

### 2.2. Convolutional neural networks

A DNN is composed of a number of "neurons" organized into layers, with each neuron contributing to the output with a particular intermediate computation. In CNNs, the FP stage for a convolution layer (CONV) comprises a convolution operator that applies a collection of filters $F$ to an input $I$ to produce the output $O$:

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

Let us consider that $F$ consists of $k_n$ filters (or kernels) of dimension $k_h \times k_w \times c_i$, where $k_h \times k_w$ specify the dimensions of the (2D) filter and $c_i$ stands for the number of input channels.[1] Furthermore, assume the layer receives an input tensor $I$ composed of $t$ samples of dimension $h_i \times w_i \times c_i$ each; and produces an output tensor $O$ with $t$ outputs of size $h_o \times w_o \times k_n$ each. Then, each of the $k_n$ individual filters in this layer combines a (sub)tensor of the inputs, with the same dimension as the filter, to produce a single scalar value (entry) in one of the $k_n$ outputs. By repeatedly applying the filter to the whole input in a sliding window manner (and with a certain vertical/horizontal stride $s_v$ and $s_h$), the convolution operator produces the complete entries of this single output; see [30]. Assuming vertical/horizontal paddings given by $p_v$ and $p_h$, the output dimensions become $h_o = \lfloor (h_i - k_h + 2p_v)/s_v + 1 \rfloor$

[1] Unless otherwise explicitly stated, in the following algorithms and data structures we adopt a generalization of the Fortran memory storage convention for multidimensional arrays (tensors) where the entries are stored in consecutive positions in memory starting from the leftmost indices.

**Table 1**
Dimensions of tensor operands for the convolution $O = \text{CONV}(F, I)$ without and with the IM2COL transform (top and bottom, respectively). In the latter case, the convolution is performed as the GEMM $C = A \cdot B$, where $C \equiv O$, $A \equiv F$, and $B = \text{IM2COL}(I)$. In this table, for simplicity we assume that $s_v = s_h = 1$ and $p_v = p_h = 0$.

| Stage | $O$ | $F$ | $I$ |
|---|---|---|---|
| FP | $h_o \times w_o \times k_n \times t$ | $k_h \times k_w \times c_i \times k_n$ | $h_i \times w_i \times c_i \times t$ |
| BP ($\partial O/\partial I$) | $h_i \times w_i \times c_i \times t$ | $k_h \times k_w \times c_i \times k_n$ | $h_o \times w_o \times k_n \times t$ |
| BP ($\partial O/\partial F$) | $k_h \times k_w \times c_i \times k_n$ | $h_o \times w_o \times k_n \times t$ | $h_i \times w_i \times c_i \times t$ |

| Stage | $C$ | $A$ | $B$ |
|---|---|---|---|
| FP | $k_n \times (h_o \cdot w_o \cdot t)$ | $k_n \times (k_h \cdot k_w \cdot c_i)$ | $(k_h \cdot k_w \cdot c_i) \times (h_o \cdot w_o \cdot t)$ |
| BP ($\partial O/\partial I$) | $(k_h \cdot k_w \cdot c_i) \times (h_o \cdot w_o \cdot t)$ | $k_n \times (k_h \cdot k_w \cdot c_i)$ | $k_n \times (h_o \cdot w_o \cdot t)$ |
| BP ($\partial O/\partial F$) | $k_n \times (k_h \cdot k_w \cdot c_i)$ | $k_n \times (h_o \cdot w_o \cdot t)$ | $(k_h \cdot k_w \cdot c_i) \times (h_o \cdot w_o \cdot t)$ |

```
L1:  for i_t = 0, ..., t − 1
L2:    for i_c = 0, ..., c_i − 1
L3:      for i_w = 0, ..., w_o − 1
L4:        for i_h = 0, ..., h_o − 1
             c = i_h + i_w · h_i + i_t · w_i · h_i
L5:          for i_kw = 0, ..., k_w − 1
L6:            for i_kh = 0, ..., k_h − 1
                 r       = i_kh + i_kw · k_h + i_c · k_w · k_h
                 B[r][c] = I[i_h · s + i_kh][i_w · s + i_kw][i_c][i_t]
```

**Fig. 1.** Algorithm for the IM2COL transform. The actual implementation eliminates some of the loop invariants inside Loops L4 and L6 to reduce the indexing arithmetic overhead.

and $w_o = \lfloor (w_i - k_w + 2p_h)/s_h + 1 \rfloor$. The top part of Table 1 displays the dimensions of the tensor operands involved in a convolution operator in the three stages of DNN training: FP, BP-GC and BP-WU; see [7] for details.

### 2.3. Convolution operators via GEMM: the IM2COL in FP

In modern computer architectures, the performance of a direct realization of the convolution operator is constrained by the memory bandwidth, delivering only a fraction of the processor peak floating-point throughput. In practice, higher performance can be attained via an indirect (or GEMM-based) approach that casts this operator in terms of a matrix multiplication, as explained next. Concretely, the IM2COL transform [10], applied to the convolution operator appearing in the FP stage, transforms the input tensor $I$ into an augmented matrix $B$ so that the output of the application of the convolution $O = \text{CONV}(F, I)$ can be obtained from the GEMM:

$$C = A \cdot B = A \cdot \text{IM2COL}(I),$$

where $C \equiv O \rightarrow k_n \times (h_o \cdot w_o \cdot t)$ is the output tensor viewed as an $m \times n$ matrix, with $m = k_n$ and $n = h_o \cdot w_o \cdot t$; $A \equiv F \rightarrow k_n \times (k_h \cdot k_w \cdot c_i) = m \times k$ contains the filters; and $B \rightarrow (k_h \cdot k_w \cdot c_i) \times (h_o \cdot w_o \cdot t) = k \times n$ results from applying the IM2COL transform to the input tensor $I$ according to the filter dimensions and strides ($k_h$, $k_w$, $s_v$, $s_h$). Fig. 1 shows the algorithmic realization of the IM2COL transform.

Fig. 2 depicts graphically the transformation of the convolution operator into a GEMM realized via the IM2COL transform. The use of the "reshape" operator $A \equiv \text{RESHAPE}(F)$ there re-arranges the input 4D filter tensor $F$ as the 2D matrix $A$. In addition, the reshape followed by a transpose $O \equiv \text{RESHAPE}(C)^{T(1,2,0,3)}$, where the superindex $(1, 2, 0, 3)$ specifies the permutation applied to the dimensions of RESHAPE($C$), re-organizes the resulting $C$ matrix back into the 4D output tensor $O$.

In general, the FP stage of a convolution layer using the IM2COL transform can be represented as the computational graph in (the blue parts of) Fig. 3. The nodes in that graph represent the kernels (GEMM and element-wise addition), while the edges are tagged with the corresponding transforms (IM2COL, RESHAPE, and transpose). We also include the model biases ($b$), as a vector of length $k_n$.

### 2.4. The COL2IM/IM2COL in BP

The operations in BP for a convolution layer include the computation of gradients of the output with respect to the inputs ($\partial O/\partial I$), filters ($\partial O/\partial F$), and biases ($\partial O/\partial b$), as explained next:

- To compute the downstream gradient with respect to the inputs ($\partial O/\partial I$), it is necessary to "reverse" the operations in the FP stage using the upstream gradient ($\partial O/\partial O$), as shown in (red in) the top left branch of the computational graph in Fig. 3. This computation can be realized using a GEMM involving $A^T$ and $\partial O/\partial C$, followed by a COL2IM operator applied to the result:

  $$\partial O/\partial I = \text{COL2IM}(A^T \cdot \partial O/\partial C),$$

  where $\partial O/\partial I \rightarrow (h_i \times w_i \times c_i \times t)$ is the tensor resulting from the application of COL2IM (considering the parameters $k_h, k_w, s_v, s_h$) to the result of the matrix multiplication of the transposed filters $A^T \equiv F^T \rightarrow (k_h \cdot k_w \cdot c_i) \times k_n$ with the upstream gradient $\partial O/\partial C \rightarrow k_n \times (h_o \cdot w_o \cdot t)$. Basically, the COL2IM transform, given in Fig. 4, reverses the effect of IM2COL, except for the fact that COL2IM accumulates several partial results on the same 4D output coordinates.
- The downstream gradient with respect to the filters ($\partial O/\partial I$) is computed via the GEMM:

  $$\partial O/\partial F = \text{RESHAPE}(\partial O/\partial C \cdot B^T),$$

  where $\partial O/\partial F \rightarrow (k_h \times k_w \times c_i \times k_n)$ is the 4D reshaped tensor resulting from the output 2D tensor from the GEMM on the upstream gradient $\partial O/\partial C \rightarrow k_n \times (h_o \cdot w_o \cdot t)$, and $B^T \equiv \text{IM2COL}(I)^T \rightarrow (h_o \cdot w_o \cdot t) \times (k_h \cdot k_w \cdot c_i)$; see middle left graph branch in (red in) Fig. 3. Here $B$ is the same augmented matrix that resulted from applying the IM2COL transform with respect to the inputs in the FP stage. In consequence, provided all the augmented matrices assembled in the convolution layers (one per layer) during FP were stored, they can be re-used at this point by operating on their transpose. However, this requires a large amount of memory, which can easily exceed the memory capacity of a computer node for CNNs with many layers.
- Finally, the downstream gradient with respect to the biases ($\partial O/\partial b$) is straight-forward to compute by summing the rows of the (2D tensor) transformed upstream gradient $\partial O/\partial C$ (see bottom left graph branch in red in Fig. 3), yielding a bias update vector of length $k_n$.

## 3. Efficient realization of convolution operators via BLIS GEMM

The previous section exposes that the GEMM is a key kernel for the efficient realization of the convolution. In this section, we re-

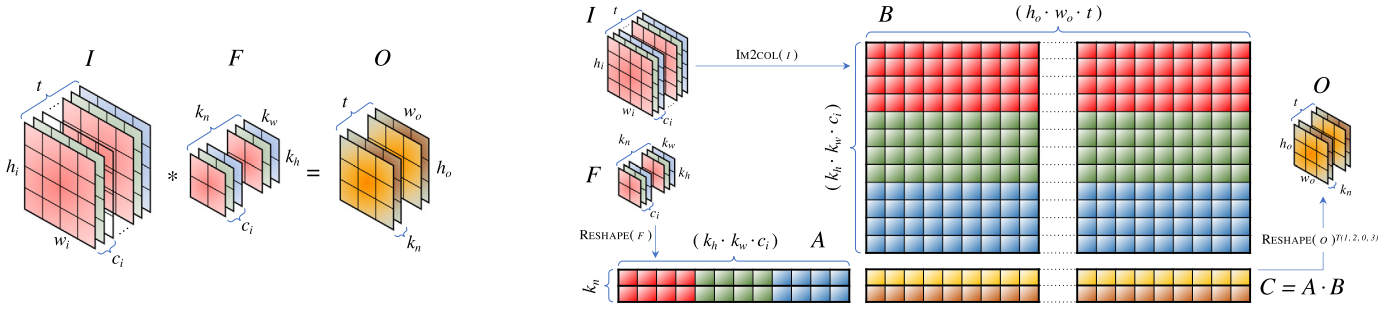


**Fig. 2.** Convolution operator via the IM2COL transform for FP.

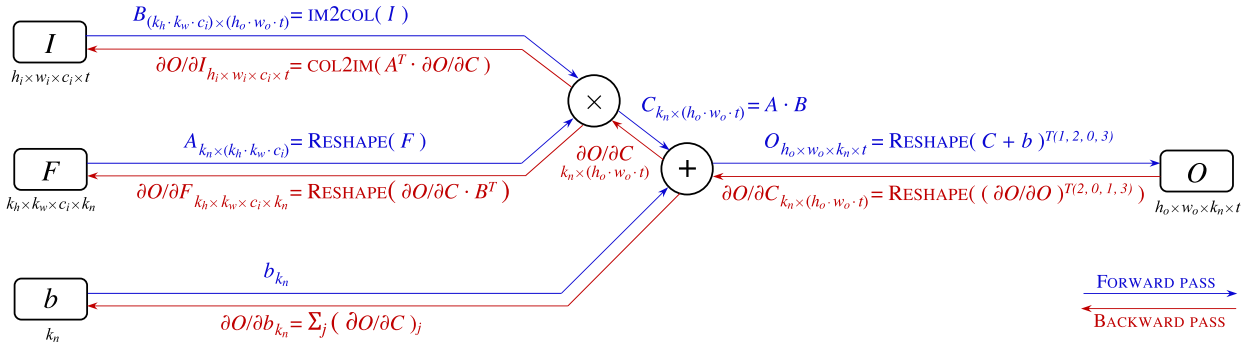


**Fig. 3.** Computational graph of the convolution layer using the IM2COL transform. From left to right, the flow denoted in blue corresponds to the FP stage. The inputs are the activations from the previous layer ($I$), the filters ($F$), and the biases ($b$). From right to left, the flow in red specifies the operations for the BP stage that distribute the upstream gradient ($\partial O/\partial O$) back to the inputs ($\partial O/\partial I$), filters ($\partial O/\partial F$), and biases ($\partial O/\partial b$).

```
L1:  for i_t = 0, ..., t − 1
L2:      for i_c = 0, ..., c_i − 1
L3:          for i_w = 0, ..., w_o − 1
L4:              for i_h = 0, ..., h_o − 1
                     c = i_h + i_w · h_i + i_t · w_i · h_i
L5:                  for i_kw = 0, ..., k_w − 1
L6:                      for i_kh = 0, ..., w_h − 1
                             r      = i_kh + i_kw · k_h + i_c · k_w · k_h
                             ∂O/∂I[i_h · s + i_kh][i_w · s + i_kw][i_c][i_t] += (A^T · ∂O/∂C)[r][c]
```

**Fig. 4.** Algorithm for the COL2IM transform. The actual implementation eliminates some of the loop invariants inside Loops L4 and L6 to reduce the indexing arithmetic overhead; for simplicity, this is not shown in the algorithm.

```
L1:  for j_c = 0, ..., n − 1 in steps of n_c
L2:      for p_c = 0, ..., k − 1 in steps of k_c
             B(p_c : p_c + k_c − 1, j_c : j_c + n_c − 1) → B_c        // Pack into B_c
L3:          for i_c = 0, ..., m − 1 in steps of m_c
                 A(i_c : i_c + m_c − 1, p_c : p_c + k_c − 1) → A_c    // Pack into A_c
                 C(i_c : i_c + m_c − 1, j_c : j_c + n_c − 1) += A_c · B_c   // Macro-kernel
```

**Fig. 5.** High performance implementation of GEMM in BLIS. $A_c$, $B_c$ 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.

view in some detail the open implementation of this kernel in the BLIS framework, as a preliminary step to illustrate how to integrate the convolution operators appearing in the FP-BP iteration inside the BLIS GEMM.

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

Consider the GEMM operation $C \mathrel{+}= A \cdot B$, where $C \to m \times n$, $A \to m \times k$, and $B \to k \times n$. BLIS implements this kernel (as well as any variant with transposed $A$ and/or $B$) as three nested loops around a *macro-kernel* plus two *packing routines*; see Fig. 5. BLIS decomposes the macro-kernel into two-three additional loops plus a micro-kernel; see [26] for details.

There are a few key points in BLIS GEMM that are worthy of being highlighted:

- The loop ordering, together with the packing routines and a proper selection of the loop strides $n_c$, $k_c$, $m_c$ that matches the processor cache configuration, orchestrates a regular pattern of data transfers through the memory hierarchy [33,21].
- All routines are encoded in plain C except for a small component inside the macro-kernel, known as the micro-kernel, which is vectorized using either architecture-dependent assembly instructions or vector intrinsics [33]. This enhances portability as porting all the BLIS library to a particular processor architecture only needs to develop an efficient realization of that component for the target processor, and to adjust the

values for $n_c$, $k_c$, and $m_c$, to the processor cache/memory configuration.

- The loops of the GEMM kernel to be parallelized can be selected at execution time. The multi-threaded parallelization of the BLIS GEMM kernel has been previously analyzed for conventional multicore processors [34], modern many-threaded architectures [29], and low-power (asymmetric) ARM-based processors in [8].
- The purpose of the packing routines is to arrange the elements 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 [17]. 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 $A_c$.)

As we argue in the following, the packing routines are particularly important for our implementation of the convolution operator.

### 3.2. Integration of the IM2COL transform in FP inside BLIS GEMM

In [26], we proposed to integrate the IM2COL transform into the packing of $B$ onto the buffer $B_c$ that occurs within BLIS.[2] For this purpose, during the execution of the GEMM kernel, the buffer $B_c$ is directly constructed from the contents of the input tensor $I$, instead of using the augmented matrix $B$, which is never explicitly assembled. *In the following, we will refer to our solution as an indirect convolution via the* CONVGEMM *operator:*

$$O = \text{CONVGEMM}(F, I).$$

This solution presents three key advantages:

- **Reduced workspace:** We avoid the use of the large workspace associated with the explicit assembly of the large augmented matrix $B \rightarrow (k_h \cdot k_w \cdot c_i) \times (h_o \cdot w_o \cdot t)$, as the only "additional" storage that is needed is the relatively small buffer for $B_c \rightarrow k_c \times n_c$, which in any case was already necessary in the GEMM kernel. (In case the augmented matrices are explicitly assembled and preserved for the BP stage, the requirements grow linearly with the number of convolution layers.) This is especially important for DNN inference, which only involves the FP stage and is often performed in memory-constrained devices, but not so much for DNN training.
- **High performance:** As argued earlier, the memory access cost associated with the packing of $B_c$ is well amortized with the flops that are performed in the innermost loops of the BLIS GEMM. Therefore, the overhead of implicitly applying the IM2COL transform can be expected to be low.
- **Portability:** The approach has the additional advantage that the only change that is needed in the BLIS GEMM is to replace the original packing routine for $B_c$ with a procedure that reads (and packs) the second input operand for the matrix multiplication directly from the input tensor. There is no need to modify the routine that performs the packing onto $A_c$. More importantly, there is no need to change the micro-kernel, which enhances the portability of our solution: the only part that is modified is encoded in C and depends on a small number of architecture-dependent parameters that are adjusted during the process of porting BLIS. The parameters that define the filter dimensions are "embedded" within the dimensions of the resulting matrix and, therefore, require no specific optimization.

[2] The BLIS-based CONVGEMM operators are available at https://gitlab.com/comtacts/convgemm, under a GNU General Public License v3.0.

```
L1:   for j_r = 0, ..., n_c − 1 in steps of n_r
          i = 0
L2:       for p_s = 0, ..., k_c − 1
              i_c = (p_c + p_s)/(k_h · k_w)
              i_kw = ((p_c + p_s) mod (k_h · k_w))/k_h
              i_kh = ((p_c + p_s) mod (k_h · k_w)) mod k_h
L3:           for j_s = 0, ..., n_r − 1
                  i_t = (j_c + j_r + j_s)/(h_o · w_o)
                  i_w = ((j_c + j_r + j_s) mod (h_o · w_o))/h_o
                  i_h = ((j_c + j_r + j_s) mod (h_o · w_o)) mod h_o
                  B_c[i][j_r] = I[i_kh + i_h · s][i_kw + i_w · s][i_c][i_t]
                  i = i + 1
```

**Fig. 6.** Algorithm for packing $I$ into $B_c$ while simultaneously applying the IM2COL transform. The indices $(p_c, j_c)$ correspond to the coordinates of the top-left entry for the block of matrix $B = \text{IM2COL}(I)$ that is packed; see Fig. 5. as well as parallelizes the outermost loop using an OpenMP `parallel for` construct.

The algorithm in Fig. 6 illustrates how to pack the corresponding entries of the input tensor $I$ into the buffer $B_c$ during the execution of the BLIS GEMM kernel in Fig. 5, while simultaneously performing the IM2COL transform. The algorithm packs the $k_c \times n_c$ block of matrix $B$ starting at row $p_c$ and column $j_c$ into the buffer $B_c$, reading the corresponding entries directly from the input tensor $I$.

### 3.3. Integration of the COL2IM transform in BP inside BLIS GEMM

The downstream gradient with respect to the inputs, $\partial O/\partial I$, can be obtained by applying the COL2IM transform to the result of multiplying $A^T$ with $\partial O/\partial C$:

$$\partial O/\partial I = \text{COL2IM}(A^T \cdot \partial O/\partial C).$$

Therefore, if explicitly built, the result of this matrix multiplication, $C' = A^T \cdot \partial O/\partial C \rightarrow (k_h \cdot k_w \cdot c_i) \times (h_o \cdot w_o \cdot t)$, would consume the same amount of memory as the augmented matrix $B$ produced by the IM2COL transform.

To tackle this, we have designed a DECONVGEMM operator that integrates the COL2IM transform into the BLIS realization of GEMM while accumulating the results in the actual output matrix:

$$\partial O/\partial I = \text{DECONVGEMM}(A^T, \partial O/\partial C),$$

without explicitly building the intermediate matrix $C'$.

Unfortunately, for the COL2IM case we cannot leverage the strategy proposed for the CONVGEMM operator, which integrated the transform inside one of the existing packing routines. The reason is that the realization of GEMM in BLIS writes its output matrix directly into memory from the microkernel, without any packing/unpacking of the data; see subsection 3.1. Therefore, to perform the COL2IM transform inside the BLIS GEMM, we had to modify this kernel to operate with an actual buffer $C_c$, of size $m \times n_c$, as well as develop an unpacking procedure that stores the data from $C_c$ onto the output 4D tensor $\partial O/\partial I$ while simultaneously performing the COL2IM transform during this process. This unpacking takes place at the end of each iteration of loop L1 in Fig. 5, once the corresponding $m \times n_c$ panel of the GEMM output matrix has been computed. The algorithm in Fig. 7 illustrates the unpacking procedure.

The main advantage of the DECONVGEMM operator lies in the memory savings that result from avoiding the explicit creation of the intermediate matrix $C'$. Concretely, the memory needed to perform this operation decreases from $(k_h \cdot k_w \cdot c_i) \times (h_o \cdot w_o \cdot t)$, when using GEMM followed by COL2IM, to the workspace $C_c \rightarrow m \times n_c =$

```
L1:  for r = 0, ..., n_c − 1
       i = 0
       i_t = (j_c + r)/(h_o · w_o)
       i_w = ((j_c + r) mod (h_o · w_o))/h_o
       i_h = ((j_c + r) mod (h_o · w_o)) mod h_o
L2:    for s = 0, ..., m − 1
         i_c = s/(k_h · k_w)
         i_kw = (s mod (k_h · k_w))/k_h
         i_kh = (s mod (k_h · k_w)) mod k_h
         ∂O/∂I[i_kh + i_h · s][i_kw + i_w · s][i_c][i_t] += C_c[i][r]
         i = i + 1
```

**Fig. 7.** Algorithm for unpacking $C_c$ into $\partial O/\partial I$ while simultaneously applying the COL2IM transform. The index $j_c$ corresponds to the first column of the block of matrix $C$ that is unpacked; see Fig. 5.

$(k_h \cdot k_w \cdot c_i) \times n_c$ for DECONVGEMM. In addition, the unpacking algorithm is encoded in C, depends only on the cache parameter $n_c$, and does not affect the coding of the micro-kernel, having thus no impact on the portability of the solution.

Regarding performance, the unpacking can introduce some overhead, depending on the convolution parameters, as this mechanism may evict some data for $A_c$ and/or $B_c$ from the cache hierarchy. A second source of performance inefficiency appears for the DECONVGEMM operator because the unpacking of $C_c$ into $\partial O/\partial I$ cannot be parallelized due to data dependencies between iterations, as different positions of $C_c$ may need to be accumulated into the same entries of $\partial O/\partial I$.[3] In summary, the utilization of the DECONVGEMM operator poses a trade-off between performance and memory consumption.

### 3.4. Integration of the IM2COL transform in BP inside BLIS GEMM

The computation of the downstream gradient with respect to the filters, $\partial O/\partial F$, requires a GEMM where the second matrix operand corresponds to the transposed result of the IM2COL transform on the input tensor $I$, that is,

$$\partial O/\partial F = \text{RESHAPE}(\partial O/\partial C \cdot B^T) = \text{RESHAPE}(\partial O/\partial C \cdot \text{IM2COL}(I)^T).$$

Although this operation resembles that performed by the CONVGEMM operator in FP, it differs in the fact that the result of IM2COL transform appears transposed in the GEMM operation. In the next two subsections, we describe two options or alternatives to deal with this.

#### 3.4.1. Re-indexing

To tackle the aforementioned variant of the CONVGEMM operator, we propose a novel "re-indexing" algorithm (REINDEX) that transforms the input 4D-tensor $I$ into $I'$ so that

$$\text{IM2COL}_{(k_h,k_w,s_v,s_h)}(I)^T \equiv \text{IM2COL}_{(k_h=h_o,k_w=w_o,s_v=h_o,s_h=w_o)}(I').$$

Concretely, the REINDEX transform re-orders the dimensions $h_i$ and $w_i$ of $I$ (that is, the rows and columns of all image channels in the batch) taking into account $s_v, s_h, h_o$, and $w_o$ to obtain $I'$; see the algorithm in Fig. 8. Fig. 9 illustrates the details of this transform using the re-indexing of an example image $I$ of size $h_i = w_i = 7$ padded with $p_v = p_h = 1$, which has to be convolved with a filter of size $k_h = k_w = 3$ using a stride of $s_v = s_h = 2$ in

[3] The overhead due to the unpacking operation depends on the convolution parameters, which are different depending on the layer and model. For many CNNs, these parameters vary significantly, and so are the case for the overheads. To quantify this overhead, we have measured the execution time of the unpack operation on the ResNet34 with Imagenet on 1 node with 1, 4, 8, and 16 threads. These overheads were 3.88%, 4.04%, 4.81%, and 5.66% of the total time, respectively.

```
L1:  for i_t = 0, ..., t − 1
L2:    for i_c = 0, ..., c_i − 1
L3:      for i_h = 0, ..., h'_o − 1
           i'_h = i_h/h_o + (i_h mod h_o) · s_v
L4:        for i_w = 0, ..., w'_o − 1
             i'_w = i_w/w_o + (i_w mod w_o) · s_h
             I'[i_t][i_c][i_h][i_w] = I[i_t][i_c][i'_h][i'_w]
```

**Fig. 8.** Algorithm for the REINDEX of $I$ to obtain $I'$ according to $s_v$, $s_h$, $h_o$, and $w_o$ in a CONV layer. This transform results into a memory-bound operation which can be accelerated using an OpenMP `parallel for` construct in loop L1, provided the batch size is large enough. This algorithm uses the C convention for multidimensional arrays: the entries of the tensors are stored in consecutive positions in memory starting from the rightmost indices.

the FP stage. As illustrated by the colored cells, the coordinates in $I$ that are two units apart ($s_v = s_h = 2$), are unraveled (and repeated) in $I'$ such that there appear $3 \times 3$ clusters of size $4 \times 4$. In consequence, this matches the number of clusters with the filter size $k_w = k_w = 3$ and the cluster size with the output dimensions $h_o = w_o = 4$. Furthermore, this allows us to leverage our CONVGEMM operator using $\partial O/\partial C$ as a filter that is convolved on the strides $s_v = h_o = 4$ and $s_h = w_o = 4$ with the transformed $I'$ to obtain $\partial O/\partial F$. In sum, the REINDEX transform combined with the CONVGEMM operator becomes:

$$\begin{aligned}\partial O/\partial F &= \text{RESHAPE}(\text{CONVGEMM}_{(s_v=h_o,s_h=w_o)}(\partial O/\partial C, I')) \\ &= \text{RESHAPE}(\text{CONVGEMM}_{(s_v=h_o,s_h=w_o)}(\partial O/\partial C, \text{REINDEX}(I))).\end{aligned}$$

It is important to note that, even though both the re-indexed tensor $I'$ and the augmented matrix $B^T \equiv \text{IM2COL}(I)^T$ require a workspace of size $(h_o \cdot w_o \cdot t) \times (k_h \cdot k_w \cdot c_i)$, the re-index algorithm has a more favorable memory access pattern than the IM2COL transform, which helps in reducing the cost of calculating $\partial O/\partial F$. Also, this solution inherits the high performance and portability advantages derived from the use of the CONVGEMM operator.

At this point, it is also worth reminding that the augmented matrices appearing in the convolution layers of the FP stage are never assembled thanks to the integration of the IM2COL transform within the BLIS realization of GEMM. In contrast, in this first option we assemble explicitly the augmented tensor $I'$ involved in the BP stage. Unlike the scenario in FP though, this matrix does not participate in subsequent layers during the BP stage and, therefore, the corresponding workspace can be re-utilized (for subsequent layers) after the GEMM operation. In practice, a single workspace of this size is affordable for most current computer nodes, especially in a training scenario using a large HPC facility.

#### 3.4.2. Transpose operand in GEMM

As an alternative option to carry out the computation of the downstream gradient with respect to the filters, $\partial O/\partial F$, we implemented a transposed version of the CONVGEMM operator such that:

$$\begin{aligned}\partial O/\partial F &= \text{RESHAPE}(\partial O/\partial C \cdot \text{IM2COL}(I)^T) \\ &= \text{CONVGEMM\_TRANS}(\partial O/\partial C, I).\end{aligned}$$

To achieve this, we developed a packing routine for the GEMM input matrix that packs $\text{IM2COL}(I)^T$ directly into $B_c$, reading the corresponding entries directly without ever assembling (or transforming) $I$ explicitly. Fig. 10 shows a simplified pseudocode of the packing routine. The algorithm packs the corresponding $n_c \times k_c$ block of matrix $B$ starting at row $j_c$ and column $p_c$.

This alternative approach based on a "transposed" CONVGEMM operator maintains all the advantages of the CONVGEMM operator

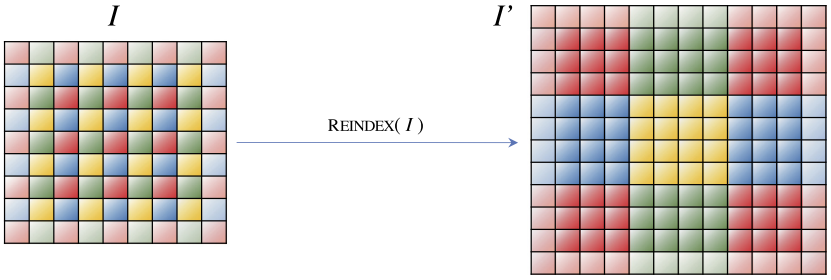


**Fig. 9.** Example of re-index of $I$ to obtain $I'$ assuming a CONV layer with $h_i = w_i = 7$, $h_o = w_o = 4$, $k_h = k_w = 3$, $p_v = p_h = 1$ and $s_v = s_h = 2$. In this example, $I$ is padded with $p_v = p_h = 1$ (see border cells) prior applying REINDEX (as these padding values were used in the FP).

```
L1:   for j_r = 0, ..., n_c − 1 in steps of n_r
          i = 0
L2:       for p_s = 0, ..., k_c − 1
              i_t = (p_c + p_s)/(h_o · w_o)
              i_w = ((p_c + p_s) mod (h_o · w_o))/h_o
              i_h = ((p_c + p_s) mod (h_o · w_o)) mod h_o
L3:           for j_s = 0, ..., n_r − 1
                  i_c = (j_c + j_r + j_s)/(k_h · k_w)
                  i_kw = ((j_c + j_r + j_s) mod (k_h · k_w))/k_h
                  i_kh = ((j_c + j_r + j_s) mod (k_h · k_w)) mod k_h
                  B_c[i][j_r] = I[i_kh + i_h · s][i_kw + i_w · s][i_c][i_t]
                  i = i + 1
```

**Fig. 10.** Algorithm for packing $I$ into $B_c$ while simultaneously applying the IM2COL transform and transposing the input matrix $I$. The indices $(j_c, p_c)$ correspond to the coordinates of the top-left entry for the block of matrix $B = \text{IM2COL}(I)^T$ that is packed; see Fig. 5. The actual realization of this algorithm eliminates some loop invariants and integer arithmetic to reduce the overhead, as well as parallelizes the outermost loop using an OpenMP `parallel for` construct.

described in Section 3.2: reduced workspace, high performance, and portability.

## 4. Integration with PyDTNN and experimental validation

In this section, we describe the integration of our new CONVGEMM-based realizations of the convolution operators in PyDTNN, a framework for distributed training of DNNs on clusters of computers. In addition, we demonstrate the benefits of the proposed approach, in comparison with one based on the explicit IM2COL/COL2IM transforms, for the distributed training of representative CNNs and datasets.

### 4.1. Overview of PyDTNN

PyDTNN[4] is a lightweight framework for distributed training of DNNs on clusters of computers that has been designed as a research-oriented tool with a low learning curve. PyDTNN presents the following appealing properties:

- **Flexible:** PyDTNN considers extensibility (and, to a certain extent, simplicity) as a first-class citizen to facilitate that users can customize the framework to prototype their research ideas.
- **Ample functionality:** PyDTNN covers DL training (and inference) for a significant part of the most common DNN models: multi-layer perceptrons (MLPs), convolutional neural networks (CNNs), residual networks (ResNets), and transformers for natural language processing. In practice, PyDTNN provides training and validation accuracies on par with those attained by Google's TensorFlow [2].
- **High performance:** PyDTNN exploits *data parallelism* (DP) [4], relying on specialized message-passing libraries for efficient communication, and kernels from high performance multi-threaded libraries for the major computational operations in CPU and GPUs. In particular, when the target cluster is equipped with NVIDIA's GPUs, PyDTNN leverages cuDNN, cuBLAS, and NCCL to deliver parallel performance that is competitive with that of TensorFlow.
- **User-amiable interface:** PyDTNN is developed in Python and offers an interface akin to that exposed by popular DL packages such as Keras.

While we recognize that PyDTNN lacks the level of maturity and the complete functionality of production-level frameworks, such as TensorFlow or PyTorch, we honestly believe that PyDTNN offers a more accessible and easier-to-customize solution, which allows us to integrate and validate the benefits of our proposed approach in the distributed training of some state-of-the-art CNNs. In any case, the proposed algorithms are orthogonal to the training framework and, as part of future work, we plan to integrate them in more sophisticated frameworks.

In a previous work [2], we demonstrated that PyDTNN delivers training/validation accuracy as well as convergence rates similar to those attained by TensorFlow. The techniques, operators, and transforms introduced in this paper do not modify the arithmetic operations performed by the convolution operators when performed via the explicit IM2COL/COL2IM transforms, only the order in which they are carried out. The small differences in the rounding errors of individual convolutions, due to this distinct order, may render slight variations in the training/validation accuracy and the convergence rate for an iterative process based on SGD (especially when combined with non-linear functions such as ReLU). We note, however, that these differences do not imply that the accuracy/convergence metrics offer necessarily worse (nor better) results.

### 4.2. Distributed training of DNNs

Training a DNN is a costly process that is usually performed on distributed high performance platforms. In practice, this tuning is often carried out on a cluster of computer nodes, each equipped with one or more multi-core processors (in some cases enhanced with graphics accelerators). Most distributed DNN training frameworks exploit DP, distributing the input data among the cluster nodes across the batch dimension, while replicating the DNN model in all nodes [4]. The DP scheme exhibits linear scalability with the number of nodes provided 1) the batch size can be increased linearly with the number of nodes; and 2) the full model fits in the node memory.

Distributed DP training requires a few types of (message-passing) collective communications [9,32]: First, the initial model has to be replicated in all nodes (Broadcast) before the training commences. Second, each FP-BP iteration requires a batch of

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

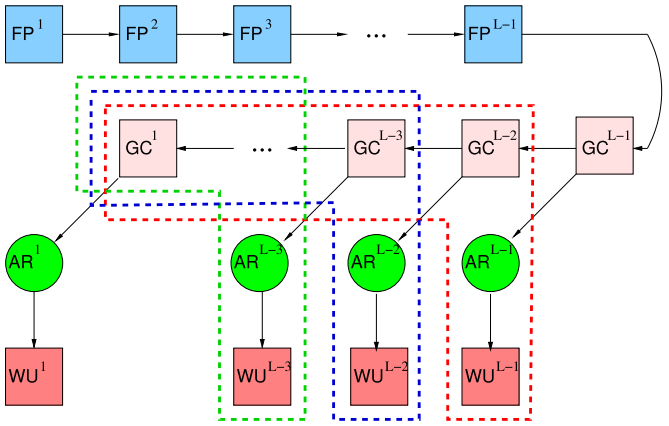


**Fig. 11.** Dependencies in the DP distributed training scheme. The colored boxes correspond to the computational stages: FP, (BP-)GC, and (BP-)WU; the circles denote the AR exchanges; and the arrows indicate data dependencies. The dashed lines of different colors identify groups of operations that can be overlapped.

samples to be distributed (Scatter) among the nodes. Third, during the BP stage, all nodes need to perform a global reduction (Allreduce or AR) of their local weights/biases onto the parameters that define the DNN model [4,7]. Given the large number of FP-BP iterations that are required to train a DNN model and the possibility of overlapping the distribution of the future batches with the FP-BP processing of the current one, the major communication bottleneck lies in the AR operation.

Fig. 11 illustrates the data dependencies and AR exchanges appearing in the DP scheme. Note the strict dependencies between adjacent layers of the FP and BP-GC stages, but the possibility of overlapping the latter with the reductions and the BP-WU computations.

### 4.3. Integration in *PyDTNN*

The efficient integration in PyDTNN of the CONVGEMM operator for the FP (and BP) stage(s) required the implementation of a new module for interfacing the CONVGEMM BLIS-based shared-library, developed in C, from Python using the ctypes library. This module allows passing Numpy arrays directly to the CONVGEMM routine so that it can be seamlessly invoked from the methods for the FP (and BP) stage(s) (for CONV layers) in PyDTNN. The versions of PyDTNN that integrate CONVGEMM also parallelize the `pad` and `transpose` methods in Numpy by means of Cython OpenMP-parallel routines, which helps to accelerate the execution. The integration of the CONVGEMM operator in the BP stage leverages the same module used for FP with extended support for the REINDEX + CONVGEMM approach, in addition to the CONVGEMM_TRANS and DECONVGEMM operators.

We note that PyDTNN uses a row-major (or C style) memory layout for multidimensional arrays while the CONVGEMM library follows the BLAS convention for column-major (or Fortran style) memory layout. Due to this, the integration also includes the necessary memory layout rearrangements to allow full compatibility between PyDTNN and the CONVGEMM library.

### 4.4. Experimental validation

#### 4.4.1. Setup

The experimental evaluation in this section has been carried out on a cluster platform consisting of 8 nodes, each equipped with two Intel Xeon Gold 5120 processors (14 cores with a nominal frequency of 2.20 GHz), and 190 GiB DDR4 non-uniform memory access (NUMA) RAM, giving each processor 95 GiB of "local" memory. The nodes are interconnected via an Infiniband EDR network with a bandwidth of 100 Gbps. Regarding the software layer, we use Intel Python 3.7.4 to run PyDTNN configured to use Numpy on top of BLIS (version 0.8.0) along with the novel CONVGEMM and DECONVGEMM routines for the Convolution proposed in this paper. The communication layer used to exploit DP is provided by Intel MPI (version 2019) configured to use the Infiniband network. For the experiments, we consider two configurations: 1) execution on a single node, exploiting multicore parallelism using multiple threads; and 2) execution using several cluster nodes with a single MPI rank and 14 threads per node. The reason to include the multi-node configurations is that varying the number of nodes modifies the relationship between computational and communication costs, which may impact the benefits of our new Convolution operators.

Training a DNN is a costly iterative procedure that often requires several epochs, each involving enough "batched" FP-BP passes to process the complete training dataset. To reduce the cost of our tests, we trained the DNN models for a fixed number of 60 FP–BP iterations, and computed the training throughput as the number of samples per batch, multiplied by 60, and divided by the time to perform the test. The fact that we perform the training for several batches, using the same FP-BP process per batch, which does not depend on the numerical values of the batch samples, has an averaging effect on the measurements. In general, for any number of nodes and threads, we observed that the throughput increased from the beginning till it stabilizes after a few seconds (iterations) due to the impact of initialization overhead. To avoid this effect, we measure the first 60 iterations to render this overhead negligible. Raising the batch size can be expected to increase linearly the execution time and memory requirements. At the same time, augmenting the batch size affects the convergence of training, often asking for an application-dependent tuning of the learning rate, which needs to be dynamically varied as the training process evolves. This is a complex technique that requires special knowledge and care, but it is out-of-scope for our work [36]. In general, we set the batch size to $t = 64$ per cluster node. We consider this is a reasonable value that offers a good balance between multi-threaded performance and small numerical distortion of the convergence rate. Increasing the batch size in general raises the cost of the explicit IM2COL/COL2IM transforms, and therefore should augment the advantage of our CONVGEMM/DECONVGEMM alternative. Also, in those cases where the memory requirements of the baseline approach exceeded the node capacity, affecting performance (due to disk swapping), the batch size was accordingly decreased. Concretely, for Resnet34 with ImageNet, the batch size was set to 24, and for DenseNet121 and GoogLeNet with ImageNet, to 16.

To analyze the parallel scalability of the proposed solutions with respect to the naive IM2COL approach, we train four representative CNN models (VGG16, ResNet34, DenseNet121, and GoogLeNet) on two datasets (CIFAR-10 and ImageNet), using the SGD optimizer and a learning rate of 0.01 for VGG16, DenseNet121, and GoogLeNet, and 0.1 for ResNet34. VGG16 is a CNN that features a 16-layer network architecture where the convolutional layer depth (number of filters) is gradually increased on a set of small ($3 \times 3$) filters [28]. ResNet34 is a 34-layer CNN belonging to the residual-based network family proposed by He et al., which introduces residual layer functions intending to ease the training of very deep CNNs [16]. Similarly, DenseNet121 is a 121-layer CNN that uses direct connections between any two layers with the same feature-map size, yielding to consistent improvements in accuracy with growing number of parameters, without signs of performance degradation or overfitting [19]. Finally, the GoogLeNet model is a 22-layer CNN that is carefully crafted to allow increasing the depth and width of the network while keeping the computational bud-

**Table 2**
PyDTNN variants for the computation of $O$ in the FP stage, and $\partial O/\partial F$, $\partial O/\partial I$ in the BP stage of a CONV layer relying on the REINDEX, CONVGEMM/CONVGEMM_TRANS, and DECONVGEMM operators.

| Variant | FP ($O$) | BP ($\partial O/\partial F$) | BP ($\partial O/\partial I$) |
|---|---|---|---|
| BASE | IM2COL + GEMM | IM2COL + transposed GEMM | GEMM + COL2IM |
| RDX | CONVGEMM | REINDEX + CONVGEMM | |
| CGT | | CONVGEMM_TRANS | |
| CGT+PM | | | |
| CGT+DG | | | DECONVGEMM |

get constant [31]. The input CONV layer in these CNNs is adapted to operate on two different datasets: *i)* the CIFAR-10 dataset, consisting of 60,000 $32 \times 32$ color images in 10 classes with 50,000 and 10,000 training and test images, respectively [20]; and *ii)* ImageNet, a dataset comprised of 1.4 M of human-annotated color photographs grouped into 1,000 classes and designed for developing computer vision algorithms [25]. In the second case, the input image size for all the networks has been downscaled to $224 \times 224$ pixels.

*4.4.2. Experiments*

In this subsection, we assess the performance of the training process using the baseline approach that explicitly builds all augmented matrices via IM2COL and COL2IM (referred to hereafter as BASE) versus four new variants that rely on the CONVGEMM/DECONVGEMM operators. All these variants leverage the CONVGEMM routine to compute $O$ in the FP stage. Furthermore, except for the last variant, they compute $\partial O/\partial I$ by invoking the BLIS GEMM followed by a call to a multi-threaded routine for COL2IM written in Cython and parallelized with OpenMP. They differ as described next:

**RDX** This implementation leverages the REINDEX transform to calculate $\partial O/\partial F$ in the BP stage; see subsection 3.4.1.

**CGT** This implementation replaces the REINDEX transform with an operator based on CONVGEMM_TRANS; see subsection 3.4.2.

**CGT+PM** This variant uses the same routines as CGT in addition to persistent memory for storing the intermediate matrices required to interface PyDTNN with the external CONVGEMM routines. This implementation aims at reducing the overhead for memory allocation/release time that are intrinsically incurred by the Python runtime (garbage collector), at the expense of yielding a higher memory footprint.

**CGT+DG** This variant leverages the same routines as CGT except that it replaces the indirect GEMM + COL2IM approach to compute $\partial O/\partial I$ by the BLIS-based DECONVGEMM routine. This alternative prevents the allocation of the augmented matrix that has to be passed to COL2IM, though it might impact performance due to the DECONVGEMM packing inefficiencies discussed in subsection 3.3.

Table 2 identifies the specific procedure to calculate $O$, $\partial O/\partial F$, $\partial O/\partial I$ for BASE and each of the four variants.

In Figs. 12 and 13, we report the performance of the training procedure using the previous realizations on a single node and on a cluster, respectively, and measured in samples/s for the selected CNNs and the two datasets. The number at the top of each bar in the performance plots specifies the speedup of the new CONVGEMM-based variants with respect to the BASE reference. Furthermore, the horizontal line and the number above the BASE bar respectively represent the peak throughput and speedup that could be achieved if the time costs related to the IM2COL transform were negligible. Both IM2COL followed by GEMM and CONVGEMM perform the same number of floating-point operations. Given that GEMM is a highly optimized operation, its execution without the overhead imposed by the IM2COL transform offers a practical theoretical bound for what we could expect to achieve with CONVGEMM if we can totally hide the overhead of the intrinsic re-organization of the data during the implicit (blocked) IM2COL that is done during packing.

Focusing on the performance results on a single node (see Fig. 12), we note that the RDX, CGT, and CGT+PM variants outperform, in most cases, the BASE approach. For ResNet34 on Imagenet with 14 threads, and for GoogLeNet on Imagenet with 8 and 14 threads, there is not enough work for all the threads, which negatively affects the performance of the BASE approach. Thus, to keep the comparison fair, we do not report the gains of the other variants for these scenarios in the next discussion. By hiding the costs of the IM2COL transform within the GEMM realization, the RDX variant improves the throughput by a factor that ranges from 1% to 11%, in most of the cases being close to the peak acceleration indicated by the horizontal line on the top of the BASE baseline. For CIFAR-10, we also detect that the CGT variant is slightly more competitive than RDX. We explain this effect due to the different loop orderings in the CONVGEMM and CONVGEMM_TRANS operators used in RDX and CGT respectively, which affect the performance depending on the input operand sizes. On the other hand, higher performance benefits can be attained by adding the persistent memory mechanism (variant CGT+PM) that prevents the Python runtime from releasing memory at execution time. In this case, the speedups range from 7% to 24%, yet they are far from yielding good strong scalability with the number of threads. (We remind that the batch size remains constant per node.) In particular, in a NUMA system, CGT+PM is only effective if the processor can access sufficient local RAM. Unfortunately, the memory requirements of DenseNet121 and GoogLeNet on ImageNet exceed the available local RAM in the target platform. In contrast, the CGT+DG variant outperforms BASE on single-threaded executions but, given the sequential unpacking (due to data dependencies) in the DECONVGEMM routine, it is only competitive on large models/datasets. However, in such cases RDX/CGT are still the recommended options.

Regarding the results in multi-node scenarios (see Fig. 13), we detect similar trends for the new CONVGEMM-based variants, with CGT+PM being the most efficient option except for DenseNet121 and GoogLeNet on ImageNet, where the highest throughput is delivered by RDX. It is also important to remark that the throughput scaling attained in the multi-node scenario is more favorable than in a single node. This is due to the linear scaling of the workload (local batch) with the number of processes/nodes (weak-scaling). In the multi-threaded scenario, the batch size remains constant while increasing the number of threads (strong-scaling), so increasing the parallel resources reports speedups (with respect to a single thread) that are in the range $[1.39, 2.79]$ for 4 threads, $[1.48, 3.72]$ for 8 threads, and $[1.51, 4.34]$ for 14 threads.

Fig. 14 shows the maximum training memory consumption of the CONVGEMM-based variants on a single node and distinct numbers of threads.[5] (The memory consumption per node for the multi-node configurations should show no differences.) For CIFAR-10, we find that RDX, CGT and CGT+DG produce memory savings ranging between 10% and 40% with respect to BASE, and that CGT is the most favorable variant. Contrarily, the use of persistent memory (CG+PM) largely exceeds the memory requirements of the BASE

[5] The memory consumption measurements were retrieved via the `getrusage` function from the `resource` Python module, which internally performs the `getrusage` POSIX system call. The routine `getrusage` was invoked twice, passing the `RUSAGE_SELF` and `RUSAGE_CHILDREN` arguments, respectively, and summing up the `ru_maxrss` statistic to account for the largest amount of physical memory occupied by the process and its children.

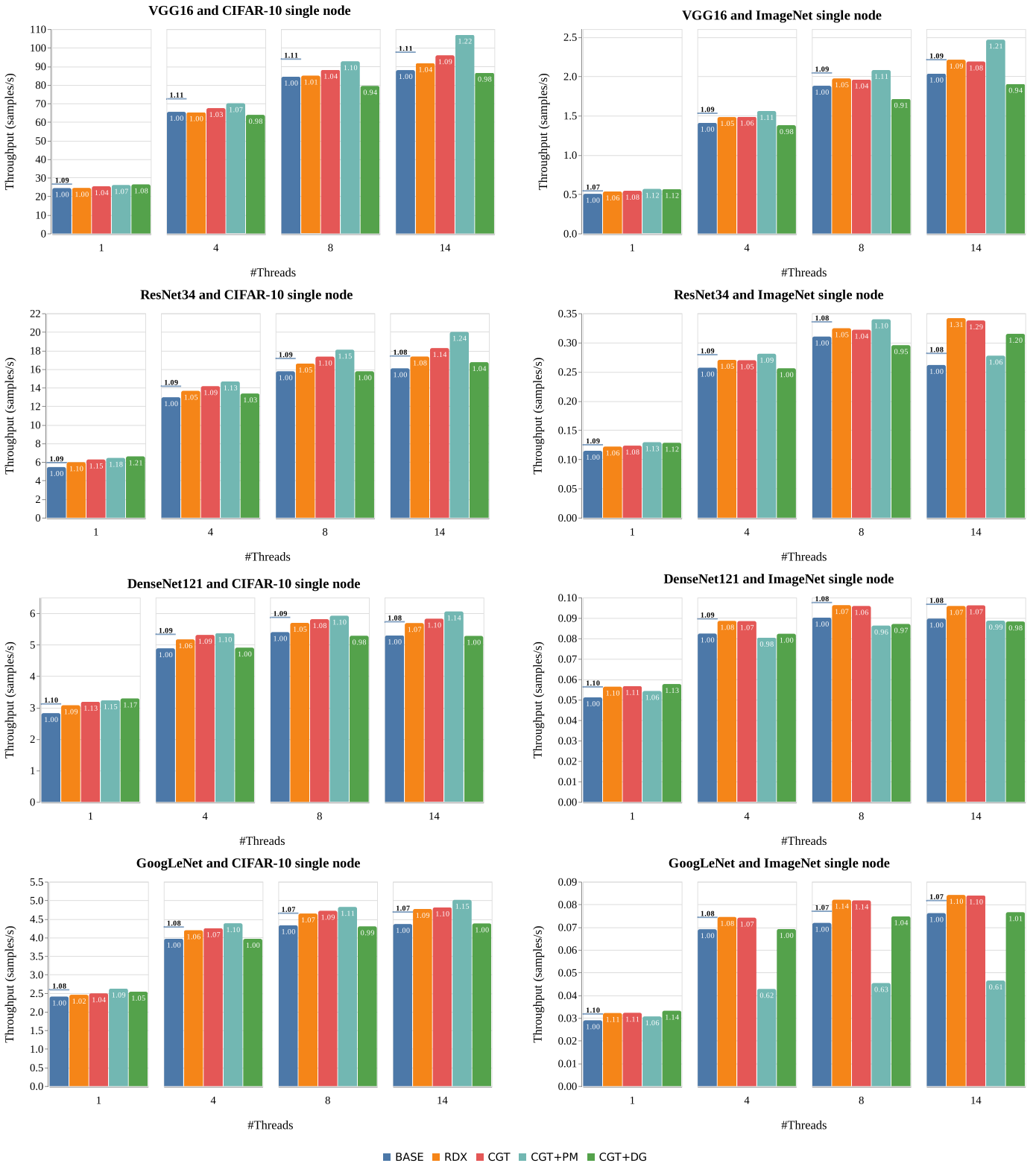


**Fig. 12.** Performance of training using a single node for the baseline approach (BASE) and the CONVGEMM-based variants with the CIFAR-10 (left) and a ImageNet (right) datasets.

reference. The savings attained by the former three variants (RDX, CGT and CGT+DG) are more evident on the dataset with the largest input size (i.e., ImageNet), though we detect larger reductions with the use of CGT+DG in nearly all the cases, with memory savings ranging between 24% and 70%. The reason for CGT+DG producing a larger memory footprint than CGT on CIFAR-10 is the memory allocation patterns appearing in the former, which lead the Python memory manager to generate a higher heap fragmentation. This effect causes the Python process to consume higher amounts of memory that are in fact not released to the OS.

All in all, we observe that the CGT+PM provides the highest speedups except when the combination of model and dataset does

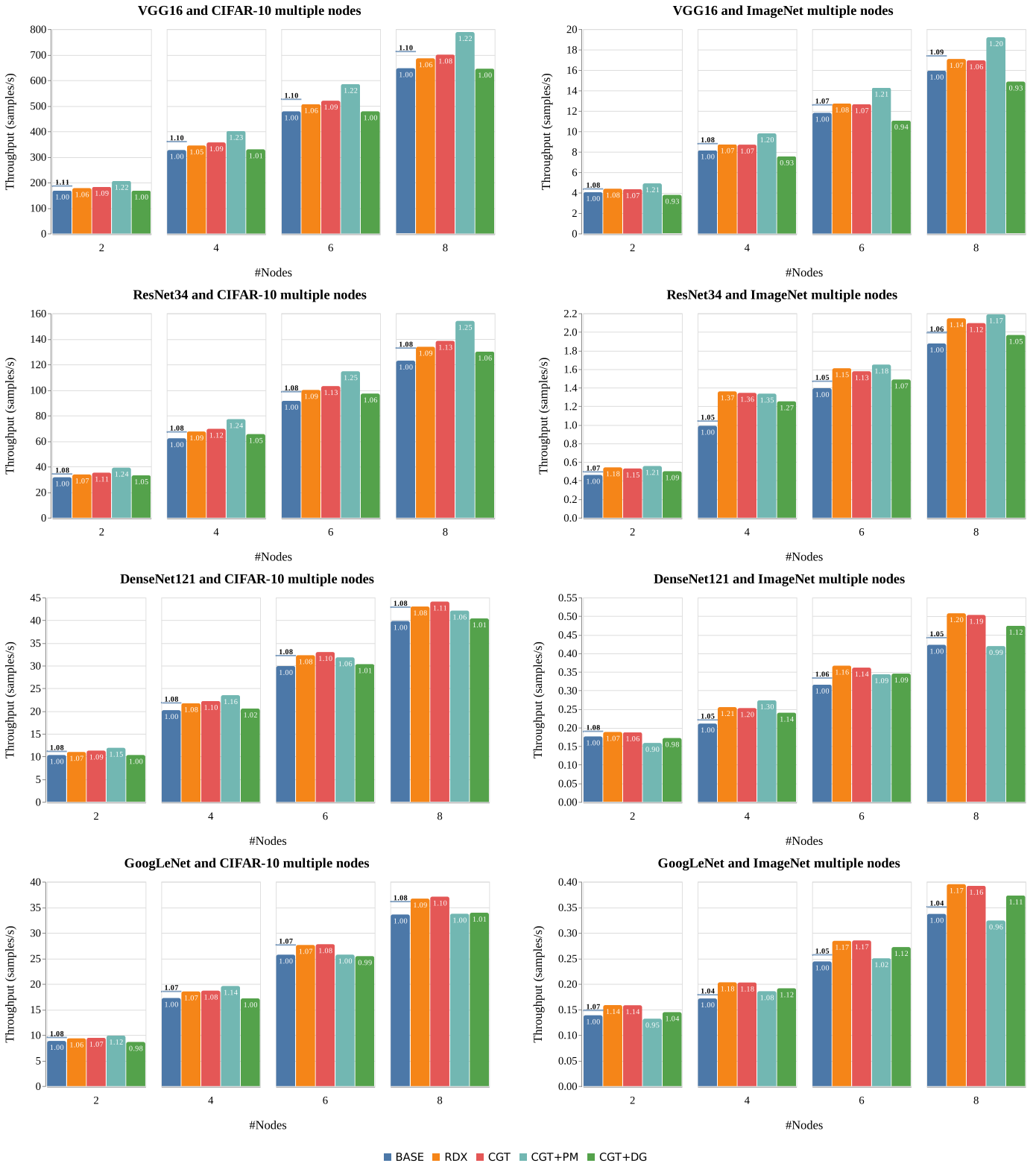


**Fig. 13.** Performance of training using a cluster for the baseline approach (BASE) and the CONVGEMM-based variants with the CIFAR-10 (left) and a ImageNet (right) datasets.

not fit into the node RAM, in which case the preferred variants are either RDX or CGT. With regards to memory consumption, CGT and CGT+DG exhibit a much lower footprint compared with BASE. Thus, we can conclude that CGT is a fair option in memory-constrained scenarios while CGT+PM guarantees higher performance at the expense of a larger memory footprint.

#### 4.4.3. Comparison with Tensorflow+Horovod

To put in perspective our results, we next review two previous works where we compared the training results achieved by PyDTNN using CPUs and GPUs against Tensorflow+Horovod.

In [2], the distributed training performance of PyDTNN was compared against TensorFlow+Horovod using the CPUs from the

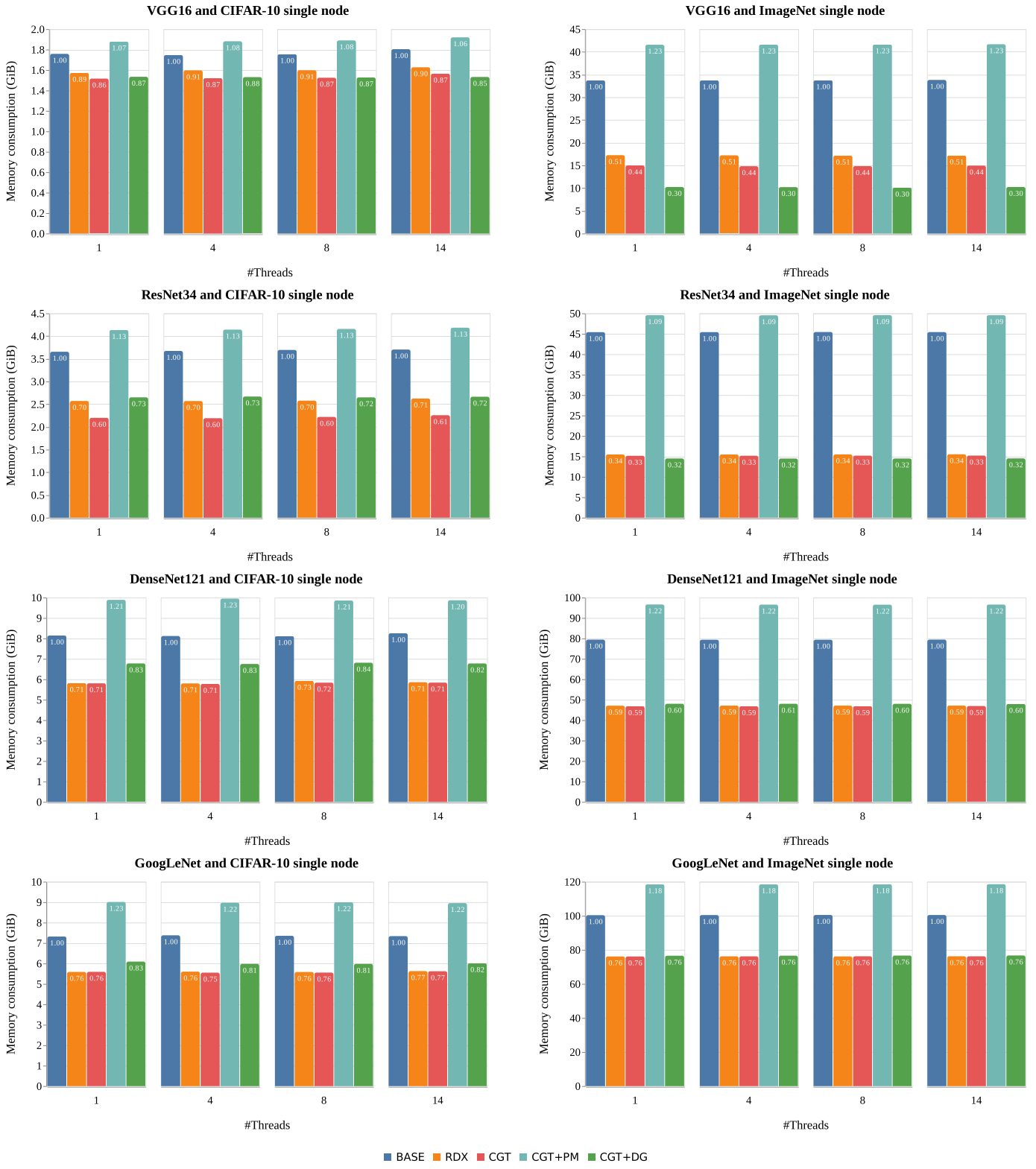


**Fig. 14.** Memory consumption of training using a single node for the baseline approach (BASE) and the CONVGEMM-based variants.

same cluster leveraged in this work. The results there illustrated that the data-parallel schema adopted by PyDTNN, with the batch size being increased linearly with the number of processes, lead to a fair weak scaling, comparable with that from TensorFlow+Horovod. The PyDTNN convolution layers in [2] were realized using the baseline algorithm described in this work, that is, using IM2COL (performed via an OpenMP-parallelized external Cython module) followed by the GEMM operation.

A second article on PyDTNN [3] provided practical evidence that the distributed training on GPUs using PyDTNN attains similar accuracy and parallel performance to those achieved by TensorFlow+Horovod on GPUs. In that case, the GPU backend of PyDTNN

was used, which internally calls the NVIDIA cuDNN library to perform the model layers related operations.

The experiments in [3] also allow a comparison between the performance on training when performed on CPUs and GPUs. Our experiments in that paper with VGG16 show a throughput difference between 60× (1 node) and 100× (8 nodes) in favor of the GPUs. This difference is not only due to the massively-parallel architecture of the GPU but also to the fact that, in the case of a cluster with GPUs, it is possible to use NVIDIA NCCL for communication between the nodes, which is more efficient than MPI.

## 5. Related work

The optimal implementation of convolution operators is an active area of research where, depending on how these operators are internally implemented, they can be classified into three groups: 1) direct convolutions; 2) indirect or GEMM-based convolutions based on the IM2COL/IM2ROW transforms; and 3) transform-based convolutions. In this section, we only review some of the solutions of the second class, as they use the same strategy as the operators proposed in this work.

With regards to solutions targeting CPUs, M. Dukhan [14] proposes an indirect convolution algorithm that avoids the overheads related to the IM2COL transform by introducing an indirection buffer, mimicking the IM2COL augmented matrix on a tailored GEMM kernel which reduces memory consumption by up to 62% according to the results. As remarked by the author, however, this solution cannot be used to perform the convolutions appearing in the backward pass and it only works for the NHWC format. Similarly, Anderson et al. [1] introduce a collection of indirect low-memory convolutions for the inference stage which match the performance of the best-known approaches, though in some cases, they require a small fraction of the additional memory. The authors in [12] present MEC, a memory-efficient convolution algorithm for deep learning, which leverages a lowering scheme to improve memory efficiency and computational efficiency for reduced memory footprint. The experimental results on different mobile and server platforms show that MEC reduces memory consumption significantly and speeds up the performance compared with other state-of-the-art solutions.

GEMM-based convolutions have also been developed for accelerators. For instance, the work by Chetlur et al. [11] lays the foundations of the cuDNN convolution routines on GPUs. Their GEMM-based convolutions use sub-tiles of the column matrix in on-chip memory, matching the sub-matrix tile size to the tile size used by the underlying GEMM implementation. They find that this strategy achieves speedups over Caffe's standard IM2COL between 0% and 30%. Other works, such as that by Zhou et al. [37], propose the memory-efficient and hardware-friendly implicit IM2COL algorithm for the Google's TPU, which dynamically converts a convolution into a GEMM with practically zero performance and memory overhead, showing as well that the algorithm can also be generally applied to NVIDIA's Tensor Cores.

Although all these proposals have the same goals as the operators presented in this work, none of them integrates the IM2COL transform within the internals of a high-performance open source realization of GEMM. By integrating them with BLIS, we inherit the performance benefits of the packing/tiling strategies for the realization of the GEMM kernel, while ensuring their portability due to the availability of BLIS micro-kernels optimized for several processor architectures.

In addition, most of the previously-cited works only target the inference stage, while our approach extends the convolution operators to the training phase as well.

## 6. Concluding remarks

In this work, we have introduced several new CONVGEMM-based operators, as well as a REINDEX transform, to compute the outputs and the downstream gradients associated with convolutional layers that are necessary to train a CNN. These operators integrate the IM2COL/COL2IM transforms within the BLIS realization of the GEMM, avoiding the creation of large intermediate matrices and a significant fraction of the time overhead. Furthermore the proposed REINDEX transform allows re-using the CONVGEMM operator in the computation of the gradient with respect to the filters. Thirdly, we have demonstrated the benefits of these new operators/transform by integrated them into PyDTNN, a simple yet efficient Python framework for distributed training of DNNs.

In more detail, our experimental evaluation using two representative CNNs and datasets reports the performance advantages and memory savings that the proposed operators bring to the PyDTNN framework over the baseline approach that relies on the explicit IM2COL and COL2IM transforms. This evaluation includes single-node and multi-node configurations, exploiting multi-threaded parallelism inside each node via the BLIS GEMM in both cases, in addition to distributed data parallelism in the second scenario. For the new CONVGEMM-based variants, on the one hand the results demonstrate that the use of pre-allocated memory along with the parallelization of some memory-bound Numpy routines accelerates training by a factor of about 6%–25% with respect to the baseline implementation. On the other hand, the use of persistent memory leads to higher memory footprints, but this is avoided by the variants using the CONVGEMM (+DECONVGEMM), which report memory savings of up to 70% (with respect to the baseline implementation). These configurations, however, produce smaller performance advantages given the intrinsic sequential nature of the DECONVGEMM operator while carrying out the COL2IM transform.

As future work, we plan to test our operators using mixed precision (FP16+FP32) for the DNN weight/bias parameters. We also plan to improve the performance of the CONVGEMM-based operators via multi-level parallelism across the GEMM-related loops, as implemented in the native BLIS GEMM.

## CRediT authorship contribution statement

**Sergio Barrachina:** Methodology, Software. **Manuel F. Dolz:** Data curation, Writing – original draft. **Pablo San Juan:** Software, Validation. **Enrique S. Quintana-Ortí:** Conceptualization, Supervision, Writing – review & editing.

## Declaration of competing interest

We declare no conflicts of interest.

## Acknowledgments

This research was funded by Project PID2020-113656RB-C21/C22 supported by MCIN/AEI/10.13039/501100011033 and Prometeo/2019/109 of the *Generalitat Valenciana*. Manuel F. Dolz was also supported by the Plan Gen–T grant CDEIGENT/2018/014 of the *Generalitat Valenciana*.

## References

[1] A. Anderson, A. Vasudevan, C. Keane, D. Gregg, High-performance low-memory lowering: Gemm-based algorithms for dnn convolution, in: 2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), 2020, pp. 99–106.

[2] 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), https://doi.org/10.1007/s11227-021-03673-z.
[3] S. Barrachina, A. Castelló, M. Catalán, M.F. Dolz, J.I. Mestre, A flexible research-oriented framework for distributed training of deep neural networks, in: IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPS Workshops 2021, Portland, OR, USA, June 17-21, 2021, IEEE, 2021, pp. 730–739.
[4] T. Ben-Nun, T. Hoefler, Demystifying parallel and distributed deep learning: an in-depth concurrency analysis, ACM Comput. Surv. 52 (4) (2019) 65, https://doi.org/10.1145/3320060.
[5] OpenAI: C. Berner, G. Brockman, B. Chan, V. Cheung, P. Debiak, C. Dennison, D. Farhi, Q. Fischer, S. Hashme, C. Hesse, R. Józefowicz, S. Gray, C. Olsson, J. Pachocki, M. Petrov, H.P. de Oliveira Pinto, J. Raiman, T. Salimans, J. Schlatter, J. Schneider, S. Sidor, I. Sutskever, J. Tang, F. Wolski, S. Zhang, Dota 2 with large scale deep reinforcement learning, arXiv:1912.06680, 2019.
[6] T.B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D.M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, D. Amodei, Language models are few-shot learners, arXiv:2005.14165, 2020.
[7] A. Castelló, M.F. Dolz, E.S. Quintana-Ortí, J. Duato, Theoretical scalability analysis of distributed deep convolutional neural networks, in: 2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID), 2019, pp. 534–541.
[8] S. Catalán, F.D. Igual, R. Mayo, R. Rodríguez-Sánchez, E.S. Quintana-Ortí, Architecture-aware configuration and scheduling of matrix multiplication on asymmetric multicore processors, Clust. Comput. 19 (3) (2016) 1037–1051.
[9] E. Chan, M. Heimlich, A. Purkayastha, R. van de Geijn, Collective communication: theory, practice, and experience, Concurr. Comput., Pract. Exp. 19 (13) (2007) 1749–1783, https://doi.org/10.1002/cpe.v19:13.
[10] K. Chellapilla, S. Puri, P. Simard, High performance convolutional neural networks for document processing, in: International Workshop on Frontiers in Handwriting Recognition, 2006, available as INRIA report INRIA-00112631 from https://hal.inria.fr/inria-001126.
[11] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, E. Shelhamer, cudnn: Efficient primitives for deep learning, CoRR, arXiv:1410.0759, 2014, http://arxiv.org/abs/1410.0759.
[12] M. Cho, D. Brand, MEC: memory-efficient convolution for deep neural network, in: Proceedings of 34th Int. Conference on Machine Learning – PMLR, Vol. 70 of ICML'17, JMLR.org, 2017, pp. 815–824.
[13] J. Devlin, M.-W. Chang, K. Lee, K. Toutanova, BERT: pre-training of deep bidirectional transformers for language understanding, arXiv:1810.04805, 2019.
[14] M. Dukhan, The indirect convolution algorithm, CoRR, arXiv:1907.02129, 2019, http://arxiv.org/abs/1907.02129.
[15] K. Hazelwood, S. Bird, D. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, A. Kalro, J. Law, K. Lee, J. Lu, P. Noordhuis, M. Smelyanskiy, L. Xiong, X. Wang, Applied machine learning at Facebook: a datacenter infrastructure perspective, in: 2018 IEEE International Symposium on High Performance Computer Architecture (HPCA), 2018, pp. 620–629.
[16] K. He, X. Zhang, S. Ren, J. Sun, Deep residual learning for image recognition, arXiv:1512.03385, 2015.
[17] G. Henry, BLAS based on block data structures, Theory Center Technical Report CTC92TR89, Advanced Computing Research Institute, Cornell University, 1992.
[18] C.F. Higham, D.J. Higham, Deep learning: an introduction for applied mathematicians, arXiv:1801.05894, 2018.
[19] G. Huang, Z. Liu, K.Q. Weinberger, Densely connected convolutional networks, CoRR, arXiv:1608.06993, 2016, http://arxiv.org/abs/1608.06993.
[20] A. Krizhevsky, Learning multiple layers of features from tiny images, Tech. rep, Canadian Institute for Advanced Research, 2009, http://www.cs.toronto.edu/~kriz/cifar.html.
[21] T.M. Low, F.D. Igual, T.M. Smith, E.S. Quintana-Ortí, Analytical modeling is enough for high-performance BLIS, ACM Trans. Math. Softw. 43 (2) (2016) 12.
[22] J. Park, M. Naumov, P. Basu, S. Deng, A. Kalaiah, D. Khudia, J. Law, P. Malani, A. Malevich, S. Nadathur, J. Pino, M. Schatz, A. Sidorov, V. Sivakumar, A. Tulloch, X. Wang, Y. Wu, H. Yuen, U. Diril, D. Dzhulgakov, K. Hazelwood, B. Jia, Y. Jia, L. Qiao, V. Rao, N. Rotem, S. Yoo, M. Smelyanskiy, Deep learning inference in Facebook data centers: characterization, performance optimizations and hardware implications, arXiv:1811.09886, 2018.
[23] S. Pouyanfar, S. Sadiq, Y. Yan, H. Tian, Y. Tao, M.P. Reyes, M.-L. Shyu, S.-C. Chen, S.S. Iyengar, A survey on deep learning: algorithms, techniques, and applications, ACM Comput. Surv. 51 (5) (2018) 92, https://doi.org/10.1145/3234150.
[24] B. Pudipeddi, M. Mesmakhosroshahi, J. Xi, S. Bharadwaj, Training large neural networks with constant memory using a new execution algorithm, arXiv:2002.05645, 2020.
[25] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A.C. Berg, L. Fei-Fei, ImageNet large scale visual recognition challenge, Int. J. Comput. Vis. 115 (3) (2015) 211–252, https://doi.org/10.1007/s11263-015-0816-y.
[26] P. San Juan, A. Castelló, M.F. Dolz, P. Alonso-Jordá, E.S. Quintana-Ortí, High performance and portable convolution operators for multicore processors, in: 2020 IEEE 32nd Int. Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), 2020, pp. 91–98.
[27] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, B. Catanzaro, Megatron-LM: training multi-billion parameter language models using model parallelism, arXiv:1909.08053, 2020.
[28] K. Simonyan, A. Zisserman, Very deep convolutional networks for large-scale image recognition, arXiv:1409.1556, 2015.
[29] T.M. Smith, R. van de Geijn, M. Smelyanskiy, J.R. Hammond, F.G.V. Zee, Anatomy of high-performance many-threaded matrix multiplication, in: Proc. IEEE 28th Int. Parallel and Distributed Processing Symp., IPDPS'14, 2014, pp. 1049–1059.
[30] V. Sze, Y.-H. Chen, T.-J. Yang, J.S. Emer, Efficient processing of deep neural networks: a tutorial and survey, Proc. IEEE 105 (12) (2017) 2295–2329, https://doi.org/10.1109/JPROC.2017.2761740.
[31] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S.E. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, A. Rabinovich, Going deeper with convolutions, CoRR, arXiv:1409.4842, 2014, http://arxiv.org/abs/1409.4842.
[32] R. Thakur, R. Rabenseifner, W. Gropp, Optimization of collective communication operations in MPICH, Int. J. High Perform. Comput. Appl. 19 (1) (2005) 49–66, https://doi.org/10.1177/1094342005051521.
[33] 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.
[34] F.G. Van Zee, R.A. van de Geijn, The BLIS framework: experiments in portability, ACM Trans. Math. Softw. 42 (2) (2016) 12.
[35] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A.N. Gomez, L. Kaiser, I. Polosukhin, Attention is all you need, arXiv:1706.03762, 2017.
[36] Y. You, et al., Large-batch training for LSTM and beyond, Tech. Rep. UCB/EECS-2018-138, Electrical Engineering and Computer Sciences, University of California at Berkeley, 2018.
[37] Y. Zhou, M. Yang, C. Guo, J. Leng, Y. Liang, Q. Chen, M. Guo, Y. Zhu, Characterizing and demystifying the implicit convolution algorithm on commercial matrix-multiplication accelerators, arXiv:2110.03901, 2021.

**Sergio Barrachina** graduated in Telecommunications Engineering (major in Electronics) from the Technical University of Valencia in 1995 and earned his PhD in Computer Engineering at Jaume I University in 2003, where he has been an Associate Professor in Computer Architecture and Technology since 2012. He has been teaching mainly first- and second-year courses of the former Computing degree and the current Computer Engineering and Computational Mathematics degrees. He is a member of the High Performance Computing & Architectures (HPC&A) research group, with which he has participated in numerous projects related to high-performance computing and architectures.

**Manuel F. Dolz** received his PhD in Advanced Computer Systems at the Universitat Jaume I (Spain) in 2014 and he currently is a distinguished researcher at the same university. During his career, he worked as a pre and postdoctoral researcher at the University of Hamburg and University of Carlos III Madrid for the EU projects Exa2Green and RePhrase, respectively. Manuel has also participated in other research projects at national and regional levels. His main research interests are parallel programming environments, energy efficiency, and deep learning for the highperformance parallel computing domain. Manuel has participated in different international conferences and workshops program committees and acted as a reviewer in international conferences and scientific journals. In total, he has coauthored 80+ articles in conferences and national and international journals, 28 of them indexed in JCR.

**Pablo San Juan** is a postdoctoral researcher at the Technical University of Valencia. He finished his doctoral studies in computer science in 2018 and has been working in several research projects related to HPC since then. His PhD thesis was centered in an HPC view of the NonNegative Matrix Factorization and his latest works have been focused in low-level optimization for Deep Learning.

**Enrique S. Quintana-Orti** is Professor of Computer Architecture at Technical University of Valencia. Enrique's research pursues the optimization of numerical algorithms and deep learning frameworks for general-purpose processors as well as hardware accelerators. During the past 22 years, he has coauthored 100+ papers in peer-reviewed scientific journals and 200+ in international conferences. He has also participated in international projects funded by European organizations (EU FP7 TEXT and Exa2GREEN, EU H2020 INTERTWinE and OPRECOMP), as well as in USA projects from DoE and NSF. Enrique's research on fault-tolerance has been recognized in by USA NASA with two awards, and his contributions to the acceleration of linear algebra algorithms received the 2008 NVIDIA Professor Partnership Award. Finally, he has served as member of the scientific committee of 80+ international conferences; he is area editor for Elsevier's Parallel Computing journal and he is Associate Editor for ACM Trans. on Mathematical Software.