High Performance and Portable Convolution Operators for Multicore Processors

Pablo San Juan  
*Universitat Politècnica de València*  
Valencia, Spain  
p.sanjuan@upv.es

Adrián Castelló, Manuel F. Dolz  
*Universitat Jaume I*  
Castellón, Spain  
{adcastel,dolzm}@uji.es

Pedro Alonso-Jordá, Enrique S. Quintana-Ortí  
*Universitat Politècnica de València*  
Valencia, Spain  
palonso@upv.es, quintana@disca.upv.es

Abstract—The considerable impact of Convolutional Neural Networks on many Artificial Intelligence tasks has led to the development of various high performance algorithms for the convolution operator present in this type of networks. One of these approaches leverages the IM2COL transform followed by a general matrix multiplication (GEMM) in order to take advantage of the highly optimized realizations of the GEMM kernel in many linear algebra libraries. The main problems of this approach are 1) the large memory workspace required to host the intermediate matrices generated by the IM2COL transform; and 2) the time to perform the IM2COL transform, which is not negligible for complex neural networks. This paper presents a portable high performance convolution algorithm based on the BLIS realization of the GEMM kernel that avoids the use of the intermediate memory by taking advantage of the BLIS structure. In addition, the proposed algorithm eliminates the cost of the explicit IM2COL transform, while maintaining the portability and performance of the underlying realization of GEMM in BLIS.

Index Terms—Convolutional neural networks, high performance, multicore processors.

I. INTRODUCTION

During the past two decades, the use of deep neural networks (DNNs) for machine learning (also known as deep learning, or DL), and more specifically convolutional neural networks (CNNs), have gained tremendous momentum, carrying beyond conventional problems in image classification, object detection, speech recognition and neural machine translation [1]–[3], to be extended to a myriad of unexplored applications, for example, in quantum computing, solid state lighting, nanoelectronics and nanomechanics, high throughput screening of new materials, computer vision in microscopy, radiography and tomography, and astrophysics simulation; see [4]–[6] among many others.

Current CNN models consist of a large number of neuron layers that allow to deliver superior accuracy on many artificial intelligence (AI) tasks, at the cost of a considerable computational cost, both for training and inference [4]. This cost comes from the CNN being mostly composed of convolutional layers (CONV), each basically embedding a high-dimensional convolution operator [7].

The high computational cost of the CONV layers can be tackled via certain compression techniques (such as use of low-rank approximations, quantization/low-precision arithmetic, sparsification, etc.), which aim to reduce the complexity of the convolution in exchange for a potential degradation in accuracy [8]. The application of the convolution operator can also be accelerated via optimized implementations of this kernel that reduce its arithmetic cost and/or carefully exploit the architecture of modern high performance processors, such as multicore processors and graphics processing units (GPUs): On the one hand, when the filters involved in the convolution are of size $5 \times 5$ or larger, this kernel is usually realized via the Fast Fourier transform (FFT). On the other hand, for smaller (yet more often encountered) filters, the operator is cast in terms of a general matrix multiplication (GEMM) [9]–[11] via the IM2COL transform [12]. In some cases, the GEMM-based approach can be accelerated employing Winograd’s minimal filtering algorithms, possibly combined with the Strassen variant of the matrix multiplication [13], [14]. However, this latter strategy can also result in a decay of accuracy of the trained model. (A more complete review on related algorithms for the convolution operator is provided in the complementary technical report [15].)

High performance realizations of the convolution operator/GEMM are available in libraries such as Intel’s oneDNN/MKL [11] and NVIDIA’s cuDNN/cuBLAS [16]. Optimized convolution operators for ARM processors are provided in NNPACK (https://github.com/Maratyszcza/NNPACK) and Feather-CNN [17]. However, these implementations target Intel/AMD x86 architectures, NVIDIA GPUs, and ARM processors, and therefore, they are not portable to other architectures. Moreover, some of these libraries take a “black-box” approach and their contents cannot be
the GEMM (BLAS) [19] featuring several appealing properties:

- BLIS is written in Standard C (mainly ISO C90 with a few C99 extensions).
- The BLIS code is mostly architecture-independent and, therefore, largely portable. Developing an efficient instance of BLIS for a specific processor architecture requires an efficient implementation of a small piece of code, known as the micro-kernel, and the selection of a number of cache configuration parameters that can be adjusted via an analytical model [20].
- There exist high performance realizations of the micro-kernel (and tuned selection of the cache configuration parameters) for many different architectures, including low-power ARM-based processors [21].
- On a variety of modern multicore processors, BLIS has been shown to deliver sustained high performance [21]–[23] that rivals that of commercial libraries, such as Intel’s MKL, as well as other open high performance BLAS instances, such as GotoBLAS [24], [25], OpenBLAS [26] and ATLAS [27].

In this paper, we leverage the open implementation of the GEMM kernel in BLIS to design high performance and portable convolution operators for DL inference on general-purpose multicore processors. For this purpose, we modify the packing routines in the BLIS GEMM kernel to apply the IM2COL transform on-the-fly (that is, during the execution of the matrix multiplication) on the input tensor for the convolution operator. As a result, our approach presents the following features:

- Reduced workspace. We avoid the explicit assembly of the large-scale matrix that results from applying the IM2COL transform to the input tensor, requiring no extra workspace (other than the small buffers that are used inside the BLIS GEMM).
- High performance. Our solution mimics the performance of the BLIS GEMM, basically eliminating the overhead of the IM2COL transform, to reduce the execution time of the convolution operator to that of the associated GEMM kernel.
- Portability. The result remains as portable as BLIS since our modification of the GEMM kernel does not affect the micro-kernel nor the cache configuration parameters. In consequence our technique can be ported to any multicore architecture for which an optimized implementation of the BLIS micro-kernel exists (or can be developed).

As an additional contribution of this work, we assess the advantages of our integration of IM2COL into the BLIS GEMM by porting and evaluating the resulting convolution operator on the ARM quad-core Cortex-A57 processor (ARMv8, 64-bits) that is integrated in the NVIDIA Jetson TX2 module.

The rest of the paper is organized as follows. In Section II we describe the BLIS approach for the implementation of GEMM, briefly discussing the portability and multi-threaded parallelization of this kernel. Special attention is paid there to the packing performed within BLIS, in connection with the layout of the data in memory, as these are two keys to our approach. In Section III we review the IM2COL transform and how to leverage this function to cast a convolution in terms of the matrix multiplication. We then open Section IV with a discussion of the problems of such straight-forward scheme, proposing an alternative that embeds the IM2COL transform within the BLIS GEMM kernel, yielding a portable, high performance, integrated CONVGEMM operator for multicore processors. Finally, we evaluate the performance of the new routines on an ARM Cortex-A57 processor in Section V, and offer some final closing remarks in Section VI.

II. PORTABLE, MULTI-THREADED GEMM IN BLIS

A. General overview

Consider the GEMM operation \( C := A \cdot B \), where the dimensions of the operands are \( C \rightarrow m \times n \), \( A \rightarrow m \times k \), and \( B \rightarrow k \times n \). BLIS adheres to the high-performance taxonomy in GotoBLAS [24] to implement this kernel (and any other variant, with transposed \( A \) and/or \( B \)) as three nested loops around a macro-kernel plus two packing routines; see Loops L1–L3 in the GEMM algorithm in Figure 1. In addition, the macro-kernel is implemented in terms of two additional loops around a micro-kernel; see Loops L4 and L5 in the same figure. The micro-kernel is encoded as a loop around a rank-1 update (that is, an outer product; not explicitly shown in the figure). For simplicity, we consider hereafter 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.

In BLIS, the loop ordering, together with the packing routines and an appropriate selection of the loop strides \( n_c, k_c, m_c, n_r \) and \( m_r \) (which match the processor cache configuration), orchestrate a regular pattern of data transfers through the memory hierarchy [18], [20]. In rough detail, given a processor architecture, the goal is that a \( k_c \times n_r \) micro-panel of the buffer \( B_c \), say \( B_r \), and an \( m_r \times k_c \) micro-panel of the buffer \( A_c \), say \( A_r \),
The multi-threaded parallelization of the BLIS GEMM values for needs to develop an efficient realization of the rank–1 BLIS library to a particular processor architecture only following the BLAS convention [19], the routines for either assembly or vector intrinsics [18]. Furthermore, encoded in C except, possibly, for the rank–1 update necessary to be able to invoke the GEMM kernel.

B. Portability

An appealing property of BLIS is that all routines are encoded in C except, possibly, for the rank–1 update inside of the macro-kernel, which may be vectorized using either assembly or vector intrinsics [18]. Furthermore, following the BLAS convention [19], the routines for most other Level-3 BLAS are built on top of GEMM. This enhances portability as, given an “architecture-oblivious” instance of the BLIS GEMM, porting all the BLIS library to a particular processor architecture only needs to develop an efficient realization of the rank–1 update for the target processor, and selecting the proper values for \( n_c, k_c, m_c, n_r \), and \( m_r \) to the processor cache/memory configuration.

C. Multi-threaded parallelization

BLIS allows to choose, at execution time, which of the five loops of the GEMM kernel are parallelized. The multi-threaded parallelization of the BLIS GEMM kernel has been previously analyzed for conventional multicore processors [22], modern many-threaded architectures [23], and low-power (asymmetric) ARM-based processors in [21]. The insights gained from these experimental studies show that Loop L1 is usually a good candidate for multi-socket platforms with on-chip L3 caches; Loop L3 should be parallelized when each core has its own L2 cache; and Loops L4 and L5 are convenient choices if the cores share the L2 cache.

D. Data storage

Unless otherwise explicitly stated, hereafter we adhere to the Fortran convention that dictates the column-major order storage for matrices. This implies that, for example, the column entries of a 2D array (i.e., matrix) are arranged in consecutive positions in memory and the first element of a column follows the last element of the previous column. Note that BLAS follows the Fortran matrix storage convention and, therefore, this is necessary to be able to invoke the GEMM kernel.

E. The packing routines

The purpose of these routines is to arrange the elements of \( A \) and \( B \) into \( A_c \) and \( B_c \), respectively, so that the elements of the \( A_c \) and \( B_c \) buffers will be accessed with unit stride when executing the micro-kernel [28]. (An additional benefit of packing is that macro-panels \( A_c \) and \( B_c \) are preloaded into certain cache levels of the memory hierarchy, reducing the time to access the elements of these buffers when using them to update a micro-tile of \( C \).)

The packing routines proceed to compact the data of the input operands as follows. In the packing routine for \( A_c \), each \( m_c \times k_c \) block of \( A \) is packed into \( A_c \), with its elements organized as micro-panels of size \( m_r \times k_r \); furthermore, within each micro-panel of \( A_c \), the elements are stored in column-major order. Also, each \( k_c \times n_c \) block of \( B \) is packed into \( B_c \), with its elements arranged into micro-panels of size \( k_r \times n_r \); and each micro-panel stored into row-major order; see Figure 2.

![Figure 2: Packing in the BLIS and GotoBLAS implementations of GEMM. The arrows indicate the linear layout of the elements in the memory.](image-url)
Let us analyze the overhead introduced by the data copies necessary to perform the packing. Consider, for example, the packing for $B_c$. In principle, packing this buffer requires $k_c \cdot n_c$ memory accesses, to read the elements of matrix $B$ (from the memory) and write them into the appropriate positions of the buffer (in principle, in the L3 cache, if there is one). Each buffer is then re-utilized for the (floating-point) operations embraced by Loop L3 of the GEMM kernel (see Figure 1), which amount to $\frac{m_c}{m_w} \cdot \frac{n_c}{n_w} \cdot \frac{m_w}{m_o} \cdot \frac{n_w}{n_o} \cdot 2(m_r n_r k_c) = 2mn_kc$ flops. Thus, provided $m$ is large enough, the cost of the packing for $B_c$ is negligible compared with the amount of flops performed inside Loop L3. A similar reasoning applies to the overhead due to the packing for $A_c$.

As we will expose in the next section, the packing routines are particularly important for our implementation of the convolution operator.

### III. Convolution via Explicit IM2COL+GEMM

#### A. Convolution operator

Consider a CONV layer, appearing during inference with a CNN model, that comprises a convolution operator consisting of $k_n$ filters (or kernels) of dimension $k_h \times k_w \times c_i$ each. Assume the layer receives $b$ tensor inputs (or samples) of dimension $h_i \times w_i \times c_i$ each; and produces $b$ tensor outputs of size $h_o \times w_o \times k_n$ each. (The parameter $b$ is also often referred to as the batch size.) 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 (with a certain stride $s$), the convolution operator produces the complete entries of this single output; see [4].

Assuming a padding $p$ along dimensions $h_i$ and $w_i$, the output dimensions become $h_o = \lfloor (h_i - k_h + 2p)/s \rfloor + 1$ and $w_o = \lfloor (w_i - k_w + 2p)/s \rfloor + 1$.

The algorithm in Figure 3 provides a direct realization of a convolution operator $O = \text{CONV}(F, I)$, where $I \rightarrow h_i \times w_i \times c_i \times b$ corresponds to the input tensor, $F \rightarrow k_n \times k_h \times k_w \times c_i$ denotes the filters, and $O \rightarrow k_n \times h_o \times w_o \times b$ is the output tensor.

#### B. Tensor data storage

A tensor generalizes the concept of a matrix to that of a multidimensional array. Note though that, from the physical point of view, the tensor entries are still arranged as a linear array in memory. Here, we generalize the Fortran convention of column-major order to consider that, unless explicitly stated otherwise, the entries of the tensors are stored in consecutive positions in memory starting from the leftmost indices. This implies that, for example, if the tensor $O \rightarrow k_n \times h_o \times w_o \times b$ is stored into an 4D array $O[k_n][h_o][w_o][b]$, then its entries are consecutively arranged in memory as $O[0][0][0][0], \ldots, O[k_n - 1][0][0][0], O[0][1][0][0], \ldots, O[k_n - 1][1][0][0], \ldots, O[k_n - 1][k_h - 1][0][0], \ldots$, etc.

#### C. Indirect convolution and the IM2COL transform

On modern computer architectures, the performance of the direct realization of the convolution operator given in Figure 3 is limited by the memory bandwidth and, therefore, delivers 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 via the IM2COL transform [12]. Concretely, the algorithm in Figure 4 shows how to transform the input tensor $I$ into an augmented matrix $\hat{B}$. With this transform, the output of the application of the convolution can be obtained from the GEMM $\hat{C} = \hat{A} \cdot \hat{B}$, where $\hat{C} \equiv O \rightarrow k_n \times (h_o \cdot w_o \cdot b)$ is the output tensor (viewed as an $m \times n$ matrix, with $m = k_n$ and $n = h_o \cdot w_o \cdot b$); $\hat{A} \equiv F \rightarrow k_n \times (k_h \cdot k_w \cdot c_i)$ contains the kernels; and $\hat{B} \rightarrow (k_h \cdot k_w \cdot c_i) \times (h_o \cdot w_o \cdot b)$ results from applying the IM2COL transform to the input tensor $I$.

### IV. OPTIMIZED CONVOLUTION VIA INTEGRATION OF IM2COL INTO GEMM

There are two problems with the indirect (two-stage) procedure described in Section III that performs the con-
volution as a sequence of an explicit IM2COL transform followed by a call to the GEMM kernel:

- **P1.** Starting from an input tensor $I$ of dimension $h_i \times w_i \times c_i \times b$, the IM2COL transforms creates an augmented matrix $\hat{B}$ of size $(k_h \cdot k_w \cdot c_i) \times (h_o \cdot w_o \cdot b)$. Assuming $h_i, w_i \approx h_o, w_o$, this requires a workspace that is $k_h \cdot k_w$ times larger than the original input tensor. For current deep CNNs, even when using small $3 \times 3$ filters, this can easily exceed the memory capacity of the system.

- **P2.** On modern high performance processors, when using a realization of the GEMM kernel that is highly optimized, the overhead due to the copy and replication required by the IM2COL transform in general becomes “visible” and reduces the performance of the global (explicit) IM2COL+GEMM.

To tackle both problems, we propose a solution that integrates the IM2COL transform into the packing of $\hat{B}$ onto the buffer $B_c$. Concretely, during the execution of the GEMM kernel, the buffer $B_c$ is directly assembled from the contents of the input tensor $I$ instead of using the augmented matrix $\hat{B}$, which is never assembled. In the following, we will refer to our solution as an indirect convolution via a CONVGEMM operator. We can now justify the contributions listed in the introduction of this work (see Section I):

- Reduced workspace. We avoid the use of the large workspace present in the two-step procedure (problem P1), as the only “additional” storage that is needed is the buffer for $B_c$, which is already necessary in the BLIS GEMM kernel.

- High performance. As argued during the discussion of the packing in Section II, the memory access costs introduced by the packing of $B_c$ is well amortized with the flops that are performed in the innermost loops. Therefore, the overhead can be considered negligible (problem P2).

- Portability. The approach has the additional advantage that the only change that is needed in BLIS GEMM is to replace the original packing routine with a procedure that reads (and packs) the second input operand to the matrix multiplication directly from the input tensor. There is no need to modify the routine that performs the packing with $\hat{A}$. 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.

The algorithm in Figure 5 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 Figure 1 while, simultaneously, performing the implicit IM2COL transform. The algorithm packs the $k_c \times n_c$ block of matrix $\hat{B}$ starting at row $p_c$ and column $k_c$ into the buffer $B_c$, reading the corresponding entries directly from the input tensor $I$. As a result, the output matrix comprises the sought-after convolution:

$$O = \text{CONVGEMM}(F, I) \equiv \hat{C} = \hat{A} \cdot \hat{B} \equiv \hat{C} = \hat{A} \cdot \text{IM2COL}(I)$$

where $\hat{C} \equiv O$ and $\hat{A} \equiv F$. The actual realization of this algorithm eliminates some loop invariants and integer arithmetic to reduce the overhead, though the algorithm is shown in this basic form to improve readability.

V. PERFORMANCE EVALUATION

In this section, we assess the performance of our CONVGEMM approach (that is, with IM2COL integrated into the BLIS GEMM) against the baseline counterpart that explicitly assembles the extended input activation matrix and then performs the augmented GEMM. For this evaluation we target a high performance ARM processor present in a low-power embedded system, and perform the analysis by simulating the inference stage of three representative state-of-the-art CNNs. The source codes employed for the evaluation, including the CONVGEMM implementation, is publicly available in [29].

A. Configuration

The evaluation presented in this paper was conducted on an NVIDIA Jetson TX2 [30] platform, which integrates an ARM quad-core Cortex-A57, an NVIDIA dual-core Denver, an NVIDIA 256-CUDA core Pascal GPU,
and 8 GiB of main memory. The results reported next were obtained in the ARM Cortex-A57 only, due to the wide spread of this architecture and the availability of optimized high performance linear algebra libraries for this architecture. On the software side, the experiments employed the Ubuntu Linux distribution 18.04.4, the GNU compiler gcc 7.5.0, and BLIS 0.6.0.

As the evaluation targets inference with CNNs, all the experiments employed (IEEE) simple precision arithmetic. In general, the inference process does not benefit from the use of double precision arithmetic, and a reduced precision format (floating-point single or half; or even fixed point) is often preferred in order to improve performance and/or reduce energy consumption. BLIS provides a single-precision instance of the BLAS optimized for the ARM Cortex-A57 which features an optimized micro-kernel with $m_r \times n_r = 8 \times 12$, and sets the following cache configuration values: $n_c = 3072$, $k_c = 640$, and $m_c = 120$. The GEMM realization parallelizes loop L4 of Figure 1 and the outermost loop of the packing of $A$ using OpenMP [31]. For the packing of $B$, in CONVGEMM we also parallelize loop L1 of Figure 5; the counterpart with an explicit IM2COL parallelizes loop L2 of Figure 4.

B. Inference simulator

For the evaluation, we employ a simulator that performs the major computational stages of the convolutional layers encountered during the inference of CNN models. For the baseline case, we emulate this behavior by executing a sequence of explicit IM2COL+GEMM pairs, of the dimensions appearing in consecutive layers of the neural network. Our optimized alternative instead executes the specialized CONVGEMM kernel (of the dimensions dictated by the CNN model). The simulator allocates memory buffers for all required matrices and performs a full model evaluation for each batch size in the specified range. During inference, the output of a certain layer is basically the input data of the next layer. Our code mimics this behavior by using buffer swapping. In this way, we simulate more accurately the real data movements that take place across the cache hierarchy during the inference stage. The simulator discards some minor operations that appear in a CNN, such as the application of the nonlinear function, or the pooling/dropout layers. However, the contribution of these components to the total cost is minor.

The simulator repeatedly executes the computational operations until a certain time threshold is reached, and then divides the total wall-time by the number of repetitions to avoid noise variability in the measurements.

C. CNN Models

We have applied the simulator to study the benefits of the optimized indirect CONVGEMM algorithm using three representative CNN models: AlexNet [2], VGG16 [32], and ResNet50 [33]. The former model was selected because of its simplicity, which facilitates an easier interpretation of the results. The remaining two models were chosen because of their more complex structures and notable computational requirements. The number of convolutional layers of each model is: 5 for AlexNet, 53 for ResNet50, and 13 for VGG16. In addition, each model needs a certain amount of extra memory workspace for the explicit IM2COL transform, which depends on the batch size $b$ and adds up to 15.87 $b$ MiB for AlexNet, 13.05 $b$ MiB for ResNet50, and 110.25 $b$ MiB for VGG16. This extra workspace represents the maximum memory needed to host the largest intermediate matrix assembled by the explicit IM2COL transform when executing each model. This is a key parameter because it constrains the use of the explicit IM2COL+GEMM approach for many CNN model+platform pairs due to insufficient memory capacity. Remember that our optimized algorithm with CONVGEMM saves all this extra space by avoiding the explicit creation of the intermediate matrices.

D. Experimental results

In this subsection we report the results obtained with the simulator applied to mimic the inference process for the three selected CNN models. In these experiments, we compare the execution time of the models with either 1) an IM2COL operation followed by the GEMM on the augmented matrix (explicit IM2COL+GEMM); or 2) an IM2COL performed on-the-fly with the GEMM (referred to as CONVGEMM). To better understand the source of the observed differences, in the comparison we also include 3) the cost of the GEMM operations without (the overhead caused by) the IM2COL transforms; and 4) the separate cost of the latter. Note that, as our ultimate goal is to hide completely the cost of the IM2COL transform inside the GEMM operation, the objective for our CONVGEMM routine is to match the execution time/performance rate of the stand-alone GEMM kernel.

Figure 6 displays the execution time per input sample attained for a range of batch sizes and the three CNN models. We only show there the execution time obtained with 4 cores (a more complete evaluation is provided in complementary technical report [15]). Note that, for the AlexNet and ResNet50 models, the experiments are run up to a batch size $b = 80$, while for VGG16 the largest

\[1\] The models adhere to the specifications defined in Google’s TensorFlow benchmarks suite.
The value for this parameter is only $b = 72$. This is due to the large amount of memory required by the buffer used to hold the intermediate matrices assembled by the IM2COL transform, which exceeds the memory capacity of the device (8 GiB) when $b = 80$ for the later model.

The results in Figure 6 demonstrate that our CONVGEMM realization, with an integrated IM2COL, fully hides the cost of this transform for the AlexNet network, delivering the same execution time observed when executing only the GEMM operations. When we tackle the two remaining (more complex) CNN models, the cost of the optimized algorithm still remains close to those of the stand-alone GEMM operation while clearly outperforming the explicit IM2COL+GEMM counterpart.

Figure 7: Execution time per layer obtained by the indirect convolution algorithms for VGG16 using all four ARM Cortex-A57 cores and a batch size $b = 32$.

There is a particular case worth of being discussed in some detail. Concretely, for the explicit IM2COL+GEMM approach, Figure 6 shows a notorious decrease in performance for the VGG16 when $b > 48$. This decline is caused by the large size of the intermediate matrices (see Sec. V-C), which results in I/O swapping to disk.

The observed negative effect in performance for large batch sizes and complex network models demonstrates that the optimized CONVGEMM algorithm, with an embedded IM2COL, not only allows to perform the inference process for network models that cannot be tackled by the explicit IM2COL+GEMM, but also avoids the efficiency pitfalls due to the earlier use of disk I/O in that approach.

Figure 7 reports the execution time to compute the convolutions required at each CNN layer in the VGG16 models. The plots there illustrate that the time required per layer significantly varies between different layers. To close the experimental analysis, we note that, when using the explicit IM2COL+GEMM approach, the total memory needed for the full VGG16 model varies between 1,134 MiB and 10,701 MiB for batch size $b=1$ and 72, respectively. In contrast, the optimized CONVGEMM algorithm reduces these requirements to 1,023 MiB for $b=1$ and only 2,763 MiB for $b=72$. This corresponds to savings between 9.72% and 74.2%.

VI. CLOSING REMARKS

This work introduces a new convolution algorithm that outperforms the straight-forward IM2COL+GEMM approach in several aspects. First, the new CONVGEMM algorithm removes the need of the additional memory work space utilized by the IM2COL+GEMM approach, enabling efficient inference with large CNN models in platforms with limited memory capacity. In addition, the realization of the new scheme in combination with the BLIS kernel for GEMM yields an efficient and portable
implementation that can be migrated to any multicore architecture for which an optimized implementation of the BLIS micro-kernel exists (or can be developed).

The experimental results reported in this work, using three state-of-the-art CNNs and a representative low-power ARM-based multicore processor, show the remarkable performance advantage of the new CONVGEMM scheme, which basically eliminates the workspace and performance overheads due to the utilization of an explicit IM2COL transform.

ACKNOWLEDGMENT

This research was partially sponsored by projects TIN2017-82972-R of Ministerio de Ciencia, Innovación y Universidades and Prometeo/2019/109 of the Generalitat Valenciana. Manuel F. Dolz was also supported by the Plan GenT project CDEIGENT/2018/014 of the Generalitat Valenciana.

REFERENCES

[1] L. Deng et al., “Recent advances in deep learning for speech research at Microsoft,” in 2013 IEEE Int. Conf. Acoustics, Speech and Signal Processing, May 2013, pp. 8604–8608.

[2] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “ImageNet classification with deep convolutional neural networks,” in Proc. 25th Int. Conf. Neural Information Processing Systems - Volume 1, ser. NIPS’12, 2012, pp. 1097–1105.

[3] J. Zhang and C. Zong, “Deep neural networks in machine translation: An overview,” IEEE Intelligent Systems, vol. 30, no. 5, pp. 16–23, Sep. 2015.

[4] V. Sze et al., “Efficient processing of deep neural networks: A tutorial and survey,” Proceedings of the IEEE, vol. 105, no. 12, pp. 2295–2329, Dec. 2017.

[5] M. M. Najafabadi et al., “Deep learning applications and challenges in big data analytics,” Journal of Big Data, vol. 2, no. 1, p. 1, Feb. 2015.

[6] T. Ben-Nun and T. Hoefler, “Demystifying parallel and distributed deep learning: An in-depth concurrency analysis,” ACM Computing Surveys, vol. 52, no. 4, pp. 65:1–65:43, 2019.

[7] N. Ma et al., “ShuffleNet V2: Practical guidelines for efficient CNN architecture design,” in Proceedings European Conference on Computer Vision - ECCV 2018. Lecture Notes in Computer Science, vol. 11218, 2018, pp. 122–138.

[8] S. Han, H. Mao, and W. J. Dally, “Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding,” 2015, arXiv preprint 1510.00149. Available from https://arxiv.org/abs/1510.00149.

[9] S. Chetluru et al., “cuDNN: Efficient primitives for deep learning,” 2014, arXiv preprint 1410.0759. Available from https://arxiv.org/abs/1410.0759.

[10] E. Georganas et al., “Anatomy of high-performance deep learning convolutions on simd architectures,” in Proc. Int. Conf. for High Performance Computing, Networking, Storage, and Analysis, ser. SC ’18. IEEE Press, 2018, pp. 66:1–66:12.

[11] “oneAPI deep neural network library (oneDNN): Performance library for deep learning,” 2018, formerly known as Intel Math Kernel Library for Deep Neural Networks (Intel MKL-DNN) and Deep Neural Network Library (DNLL). Available from https://onetapi-src.github.io/oneDNN/.

[12] K. Chellapilla, S. Puri, and P. Simard, “High performance convolutional neural networks for document processing,” in International Workshop on Frontiers in Handwriting Recognition, 2006, available as INRIA-00112631 report from https://hal.inria.fr/inria-00112631.