High Performance and Portable Convolution Operators for ARM-based Multicore Processors

Pablo San Juan* Adrián Castelló† Manuel F. Dolz† Pedro Alonso-Jordá* Enrique S. Quintana-Ortí*

May 14, 2020

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.

1 Introduction

During the past decade and a half, the use of deep neural networks (DNNs) for machine learning (also known as deep learning, or DL), and more specifically convolutional neural networks (CNNs), has gained a tremendous momentum, carrying beyond conventional problems in image classification, object dectection, speech recognition and neural machine translation [11, 24, 37], 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 [33, 28, 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 [33]. This cost comes from the CNN being mostly composed of convolutional layers (CONV), each basically embedding a high-dimensional convolution operator [27].

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 [19]. The application of the convolution operator can also be accelerated

*Universitat Politècnica de València, Spain. p.sanjuan@upv.es, palonso@upv.es, quintana@disca.upv.es
†Universitat Jaume I, Castellón de la Plana, Spain. {adcastel,dolzm}@icc.uji.es

via optimized implementations of this kernel that 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 5×5 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, 15, 1] via the IM2COL transform [8]. 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 [25, 39]. However, this latter strategy can also result in a decay of accuracy of the trained model.

High performance realizations of the convolution operator/GEMM are available in libraries such as Intel's openDNN/MKL and NVIDIA's cuDNN/cuBLAS, respectively [1, 2]. However, these implementations target Intel/AMD x86 architectures and NVIDIA GPUs, and therefore, they are not portable to other architectures. Moreover, except for openDNN, these libraries take a "black-box" approach and their contents cannot be examined nor modified.

The *Basic Linear Algebra Instantiation Software* (BLIS) is a software framework for rapid development of high-performance dense linear algebra libraries [34]. BLIS implements the full functionality defined in the *Basic Linear Algebra Subprograms* (BLAS) application programming interface (API) [12] 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 an 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 [26].
- 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 [7].
- On a variety of modern multicore processors, BLIS has been shown to deliver sustained high performance [36, 32, 7] that rivals that of commercial libraries, such as Intel's MKL, as well as other open high performance instances of the BLAS, such as GotoBLAS [18, 17], OpenBLAS [29] and ATLAS [35].

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 one of 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 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.

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. After a survey of related work in the next subsection, in Section 2 we review 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 3 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 4 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 5, and offer some final closing remarks in Section 6.

1.1 Related work

Direct algorithms. Libraries such as NVIDIA's cuDNN, HexagonNN [23] and Xiaomi's MACE [3] include optimized direct convolution operators for the most frequently encountered filter dimensions and strides, falling back to default algorithms for other parameter values. In comparison, Intel's MKL-DNN [15] employs parameterized architecture-aware just-in-time code generators to produce direct optimized convolution routines at runtime.

NNPACK (Neural Networks PACKage) [4] also provides direct implementations of convolutional operators involving large filters (3×3 or 5×5) using either Winograd filters or FFT. NNPACK supports many popular deep learning frameworks (Caffe2, PyTorch, MXNET, etc.) and includes architecture-specific optimizations for ARMv7, ARMv8, and x86 processors.

Indirect algorithms. In contrast with the previous approach, GEMM-based algorithms reformulate the convolution in terms of a two-stage (or indirect) IM2COL+GEMM. This allows to leverage highly optimized realizations of the BLAS, which exists for almost any modern computer platform. As a result, the GEMM-based approach is now used in all major deep learning frameworks [13].

Facebook's QNNPACK (Quantized NNPACK) [14] extends NNPACK to perform computations in 8-bit fixed-point precision targeting convolution operators which cannot benefit from fast Winograd/FFT-based schemes. Similar to our approach, QNNPACK follows an indirect approach while aiming to eliminate the overhead of the IM2COL transform for matrix multiplication libraries.

A few other works have addressed the excessive memory consumption of GEMM-based algorithms by dividing the matrix multiplication into small kernels [10, 5]. However, the authors of these works do not consider the combination of their solutions with optimized, architecture-specific realizations of the GEMM kernel.

In [13], M. Dukhan tackles both the memory and performance issues of the indirect approach. Concretely, that work proposes to introduce an indirection structure of pointers to the convolution input operand optimized for the so-called NHWC layout. Unfortunately, the author recognizes that 1) the algorithm is not expected to be competitive with state-of-the-art patch-building algorithms [38] due to strided memory access; and 2) his solution has limited applicability for the backward pass of the convolution operator and the Transposed Convolution operator.

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

Figure 1: High performance implementation of GEMM in BLIS. In the code, $C_c \equiv C(i_c : i_c + m_c - 1, j_c : j_c + n_c - 1)$ is just a notation artifact, introduced to ease the presentation of the algorithm, while A_c, B_c correspond to actual buffers that are involved in data copies.

2 Portable and Multi-Threaded GEMM in BLIS

General overview. Consider the GEMM operation $C \mathrel{+}= A \cdot B$, where the dimension 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 [18] 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, and outer product; not explicitly shown in the figure). For simplicity, we will 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 [34, 26]. 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 $m_c \times k_c$ macro-panel of the buffer A_c, say A_r, are streamed into the floating-point units (FPUs) from the L1 and L2 caches, respectively; while the $k_c \times n_c$ macro-panel B_c resides in the L3 cache (if present).

Portability. An appealing property of BLIS is that all routines are encoded in C except, possibly, for the rank–1 update inside the micro-kernel, which may be vectorized using either assembly or vector intrinsics [34]. Furthermore, following the convention for BLAS [12], the routines for (almost) all other Level-3 BLAS are built on top of GEMM. This enhances portability as, given a "generic" (architecture-oblivious) instance of the BLIS GEMM, porting all the BLIS library to a particular processor architecture then 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.

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 [36], modern many-threaded architectures [32], and low-power (asymmetric) ARM-based processors in [7]. The insights

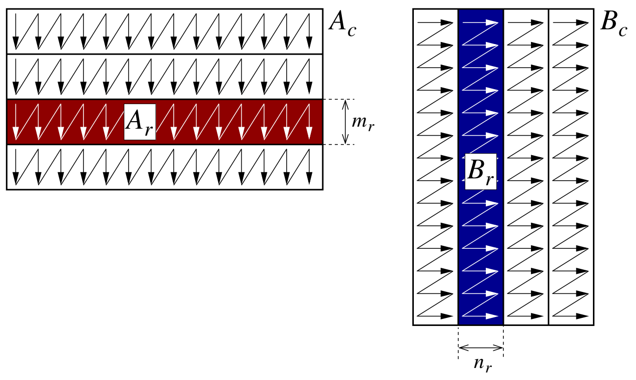


Figure 2: Packing in the BLIS and GotoBLAS implementations of GEMM. The arrows indicate the linear layout of the elements in the memory: column-major for A_r and row-major for B_r.

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.

Data storage. Hereafter, unless otherwise explicitly stated, we adhere to the Fortran convention that dictates the column-major order storage for matrices. This implies that, for example, the entries of the 2D array (i.e., matrix) $C \rightarrow m \times n$, are arranged in consecutive positions in memory as

$$\underbrace{C[0][0],\ C[1][0],\ldots,C[m-1][0]}_{\text{1st column of } C},\quad \underbrace{C[0][1],\ C[1][1],\ldots,C[m-1][1]}_{\text{2nd column of } C},\ldots,$$
$$\underbrace{C[0][n-1],\ C[1][n-1],\ldots,C[m-1][n-1]}_{\text{Last column of } C}.$$

Note that BLAS follows the Fortran matrix storage convention and, therefore, this is necessary to be able to invoke the GEMM kernel.

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 [21]. (An additional benefit of packing is that 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 copy and 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 the A_c buffer with its elements organized as micro-panels of size $m_r \times k_c$; 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 the elements arranged into micro-panels of size $k_c \times n_r$; and each micro-panel stored into row-major order; see Figure 2 and the algorithm in Figure 3.

Let us consider 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

```
L1:  for $j_r = 0, \ldots, n_c - 1$ in steps of $n_r$
       $i = 0$
L2:    for $p_s = 0, \ldots, k_c - 1$
L3:      for $j_s = 0, \ldots, n_r - 1$
           $B_c[i][j_r] = B[p_c + p_s][j_c + j_r + j_s]$
           $i = i + 1$
```

Figure 3: Algorithm for packing B into B_c. The indices p_c and j_c correspond to the coordinates of the top-left entry for the block of matrix B that is packed; see Figure 1. Matrix B is maintained in column-major order. Each micro-panel B_r within the buffer B_c is arranged in row-major order, as expected by the BLIS micro-kernel; see Figure 2. This is attained by viewing B_c as an $(k_c \cdot n_r) \times (n_c/n_r)$ buffer, where each column contains an entire micro-panel in row-major order.

```
L1:  for $i_b = 0, \ldots, b - 1$
L2:    for $i_c = 0, \ldots, c_i - 1$
L3:      for $i_w = 0, \ldots, w_o - 1$
L4:        for $i_h = 0, \ldots, h_o - 1$
L5:          for $i_{kw} = 0, \ldots, k_w - 1$
L6:            for $i_{kh} = 0, \ldots, k_h - 1$
L7:              for $i_k = 0, \ldots, k_n - 1$
                   $O[i_k][i_h][i_w][i_b] \mathrel{+}= F[i_k][i_{kh}][i_{kw}][i_c] \cdot I[s \cdot i_h + i_{kh}][i_w \cdot s + i_{kw}][i_c][i_b]$
```

Figure 4: Direct algorithm for the application of the convolution operator $O = \textsc{conv}(F, I)$.

amount to $\frac{m}{m_c} \cdot \frac{n_c}{n_r} \cdot \frac{m_c}{m_r} \cdot 2(m_r n_r k_c) = 2mn_ck_c$ 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.

3 Indirect Convolution via Explicit IM2COL+GEMM

Convolution operator. Consider a CONV layer, appearing during inference with a DNN 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 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 [33]. Assuming a padding p along dimensions h_i and w_i, the output dimensions become $h_o = \lfloor (h_i - k_h + 2p)/s + 1 \rfloor$ and $w_o = \lfloor (w_i - k_w + 2p)/s + 1 \rfloor$.

The algorithm in Figure 4 provides a *direct realization* of a convolution operator $O = \textsc{conv}(F, I)$, 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.

```
L1: for $i_b = 0,\ldots,b-1$
L2:     for $i_c = 0,\ldots,c_i-1$
L3:         for $i_w = 0,\ldots,w_o-1$
L4:             for $i_h = 0,\ldots,h_o-1$
                    $c = i_h + i_w \cdot h_i + i_b \cdot w_i \cdot h_i$
L5:                 for $i_{kw} = 0,\ldots,k_w-1$
L6:                     for $i_{kh} = 0,\ldots,w_h-1$
                            $r \quad = i_{kh} + i_{kw} \cdot k_h + i_c \cdot k_w \cdot k_h$
                            $\hat{B}[r][c] = I[i_h \cdot s + i_k h][i_w \cdot s + i_{kw}][i_c][i_b]$
```

Figure 5: Algorithm for the IM2COL transformation. The actual implementation moves 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.

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

$$\begin{array}{l}
O[0][0][0][0],\ O[1][0][0][0],\ldots,O[k_n-1][0][0][0],\\
\quad O[0][1][0][0],\ O[1][1][0][0],\ldots,O[k_n-1][1][0][0],\\
\qquad \ddots\\
\qquad\quad O[0][h_o-1][0][0],\ O[1][h_o-1][0][0],\ldots,O[k_n-1][h_o-1][0][0],\\
\qquad\qquad O[0][0][1][0],\ O[1][0][1][0],\ldots,O[k_n-1][0][1][0],\ldots,\\
\qquad\qquad\quad O[k_n-1][h_o-1][w_o-1][b-1].
\end{array}$$

Indirect convolution and the IM2COL transform. On modern computer architectures, the performance of the direct realization of the convolution operator given in Figure 4 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 [8]. Concretely, the algorithm in Figure 5 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 simply 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 $\hat{B} \rightarrow (k_h \cdot k_w \cdot c_i) \times (h_o \cdot w_o \cdot b)$ is the result from applying the IM2COL transform to the input tensor I.

4 Optimized Indirect Convolutions via Integration of IM2COL into GEMM

There are two problems with the indirect (two-stage) procedure described in Section 3 that performs the convolution 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 CNNs, with many layers, even when using small 3×3 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 +BLIS GEMM process.

To tackle both problems, we propose a solution that integrates the IM2COL transform into the packing of $\hat{B}$ onto the buffer B_c. In other words, 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 created). *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 1):

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. Furthermore, as argued during the discussion of the packing in Section 2, the memory access costs introduced by the packing of B_c is well amortized with the flops that are performed in the innermost loops and, therefore, the overhead can be considered negligible (problem **P2**).

Portability. The approach has the additional advantage that the only change that is needed to the 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 different is written 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 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 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:

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

```
L1:  for $j_r = 0, \ldots, n_c - 1$ in steps of $n_r$
         $i = 0$
L2:      for $p_s = 0, \ldots, k_c - 1$
             $i_c = (p_c + p_s)/(k_h \cdot k_w)$
             $i_{kw} = ((p_c + p_s) \bmod (k_h \cdot k_w))/k_h$
             $i_{kh} = ((p_c + p_s) \bmod (k_h \cdot k_w)) \bmod k_h$
L3:          for $j_s = 0, \ldots, n_r - 1$
                 $i_b = (j_c + j_r + j_s)/(h_o \cdot w_o)$
                 $i_w = ((j_c + j_r + j_s) \bmod (h_o \cdot w_o))/h_o$
                 $i_h = ((j_c + j_r + j_s) \bmod (h_o \cdot w_o)) \bmod h_o$
                 $B_c[i][j_r] = I[i_{kh} + i_h \cdot s][i_{kw} + i_w \cdot s][i_c][i_b]$
                 $i = i + 1$
```

Figure 6: Algorithm for packing I into B_c. The indices p_c and j_c correspond to the coordinates of the top-left entry for the block of matrix $\hat{B}$ that is packed; see Figure 1.

where $\hat{C} \equiv O$ and $\hat{A} \equiv F$. The actual implementation of this algorithm eliminates some of the loop invariants and integer arithmetic to reduce the overhead. Concretely, the computation of the indices $i_c, i_{kw}, i_{kh}, i_b, i_w, i_h$ is performed outside the loops and then properly updated during the iterations to avoid the high cost of the integer divisions and modulo operations (that is, the remainder of the integer division, abbreviated in the presentation as mod). The algorithm is shown in this basic form to improve readability.

5 Performance Evaluation

In this section, we assess the performance of our CONVGEMM approach (that is, IM2COL integrated into the BLIS GEMM) against the baseline counterpart that explicitly assembles the extended input activation matrix and then performs the augmented GEMM. As described next, 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 of all codes employed for the evaluation, including the CONVGEMM implementation, is publicly available in a git repository [16].

5.1 Configuration

The evaluation presented in this paper was executed on an NVIDIA Jetson TX2 [22] 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 processor. On the software side, the experiments were conducted using the Linux distribution Ubuntu 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:

Model	FC	CONV	POOL	Total	Memory consumption for IM2COL (MiB)
AlexNet	3	5	3	11	$15.87\,b$
ResNet50	1	53	1	55	$13.05\,b$
VGG16	3	13	5	21	$110.25\,b$

Table 1: Number and type of layers in the target CNN models and memory required by the explicit IM2COL transform as a function of the batch size b.

$n_c = 3072$, $k_c = 640$, and $m_c = 120$. The algorithm paralellizes loop L4 of Figure 1 and the outermost loop of the packing of A using OpenMP [30]. For the CONVGEMM, we also parallelize loop L1 of Figure 6. The counterpart with an explicit IM2COL parallelizes loop L2 of Figure 5.

5.2 Inference simulator

In order to tackle the complex software stack required for executing CNNs, we have employed an inference 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). In both cases, the simulator reads the CNN configuration parameters for a certain model from an input file, accepting the batch size (number of input samples simultaneously processed per inference process) as an additional parameter. The simulator then allocates memory buffers for all required matrices using the maximum size of each matrix from among the matrix sizes required by each layer in the model, 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 behaviour 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 repeatedly executes the computational operations till a certain time threshold is attained, and then divides the total wall-time by the number of repetitions to avoid system load variability in the measurements.

5.3 DNN Models

We have applied the simulator to study the benefits of the optimized indirect CONVGEMM algorithm using three representative CNN models: AlexNet [24], VGG16 [31], and ResNet50 [20].[1] 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. Table 1 summarizes the number and type of layers for each model as well as the extra memory consumption required by the explicit IM2COL transform. This later parameter represents the maximum memory needed to hold the largest intermediate matrix assembled by the explicit IM2COL transform when executing each model. This is a key parameter because it may constrain 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 this extra space by avoiding the explicit creation of the intermediate matrices.*

[1]The models adhere to the specifications defined in Google's TensorFlow benchmarks suite.

Layer	Type	Neurons $(h_i \times w_i \times c_i \times b)$	Kernels $(k_n \times k_h \times k_w \times c_i)$	GEMM dimensions $(m \times n \times k)$
2	CONV	$224 \times 224 \times 3 \times b$	$64 \times 11 \times 11 \times 3$	$64 \times 2916\,b \times 363$
4	CONV	$55 \times 55 \times 64 \times b$	$192 \times 5 \times 5 \times 64$	$192 \times 2601\,b \times 1600$
6	CONV	$27 \times 27 \times 192 \times b$	$384 \times 3 \times 3 \times 192$	$384 \times 625\,b \times 1728$
7	CONV	$13 \times 13 \times 384 \times b$	$384 \times 3 \times 3 \times 384$	$384 \times 121\,b \times 3456$
8	CONV	$13 \times 13 \times 384 \times b$	$384 \times 3 \times 3 \times 384$	$256 \times 121\,b \times 3456$

Table 2: Specification for the CONV layers appearing in the AlexNet CNN model as a function of the batch size b.

Table 2 details the configuration of the CONV layers for the AlexNet model. Concretely, the table displays the number of neurons (represented by the dimensions of the input data); the kernel specifications (number of kernels, their height and width, and their number input channels); and the dimensions of the GEMM product, when applying the indirect convolution, for each layer of that type.

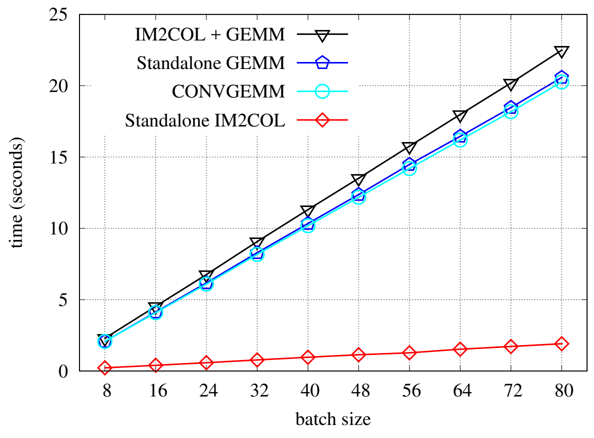
5.4 Experimental results

In this subsection we report the results obtained with the simulator applied to simulate 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 performance reference for our CONVGEMM routine is to match the execution time/performance rate of the standalone GEMM kernel.

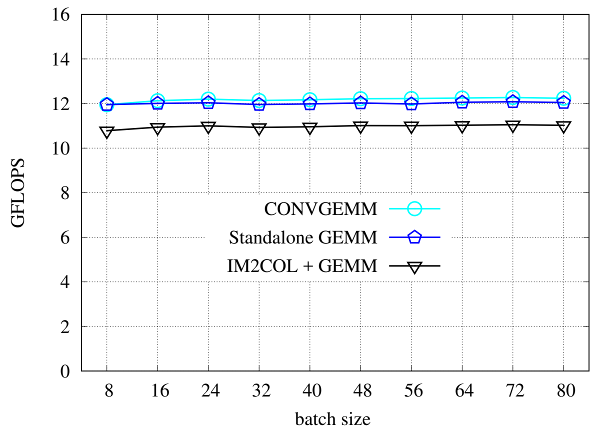
Figures 7 and 8 show the time and performance (in GFLOPS, or billions of floating-point operations per second) obtained for the evaluated models executed using a single core and the full 4-core processor, respectively. The plots display the execution time/performance attained for a range of batch sizes, for the optimized CONVGEMM algorithm against the baseline approach (explicit) IM2COL+GEMM. In addition, all plots include the execution time/performance attained by the GEMM kernels involved in the model simulation, and the plots in the left-hand side include the time overhead required to perform the IM2COL transforms.

For the AlexNet and ResNet50 models, the experiments are run up to a batch size $b = 80$, while for VGG16 the largest value for this parameter is only $b = 72$. This is due to the large amount of memory required for the intermediate matrices assembled by the IM2COL transform, which exceeds the memory capacity of the device (8 GiB) for the VGG16 model when $b = 80$.

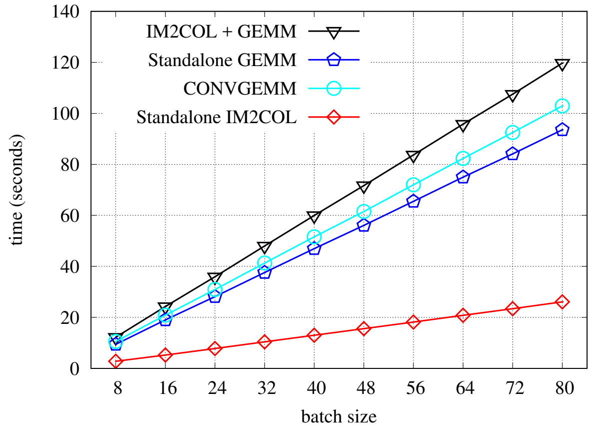
The results in Figures 7 and 8 demonstrate that our technique with an integrated IM2COL fully hides cost of this transform for the AlexNet network, delivering the same execution time and GFLOPS rate observed when executing only the GEMM operations. When we tackle the two remaining (more complex) CNN models, the cost and performance of the optimized algorithm still remain close to those of the standalone GEMM operation while clearly outperforming the explicit IM2COL+GEMM counterpart.



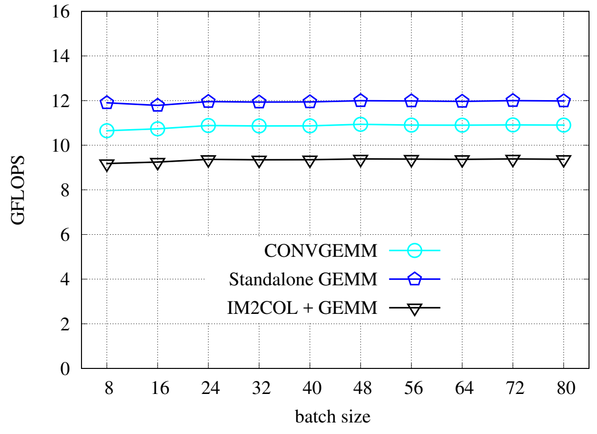
(a) Execution time, AlexNet, 1 core



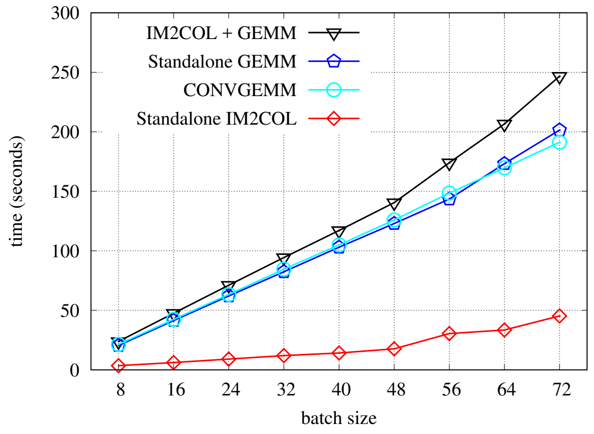
(b) GFLOPS, AlexNet, 1 core



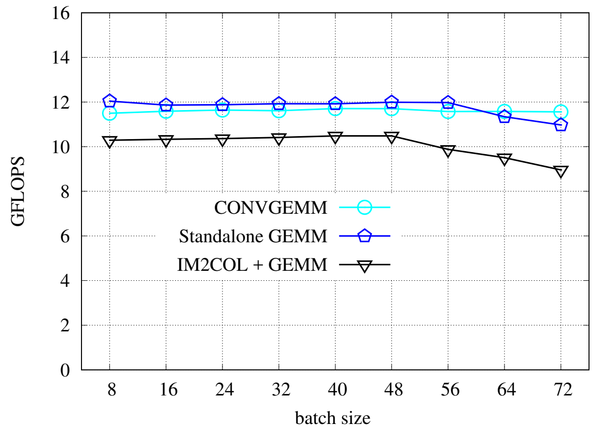
(c) Execution time, ResNet50, 1 core



(d) GFLOPS, ResNet50, 1 core

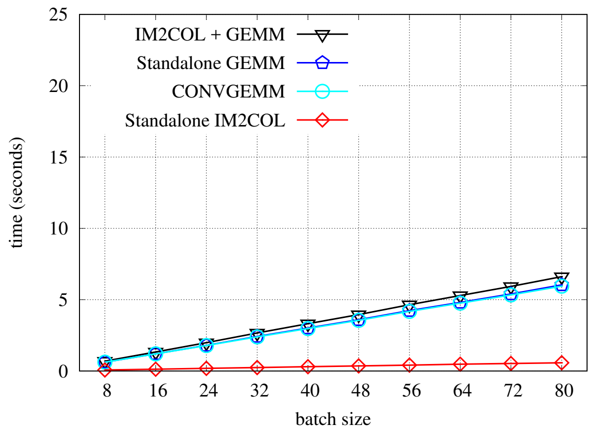


(e) Execution time, VGG16, 1 core

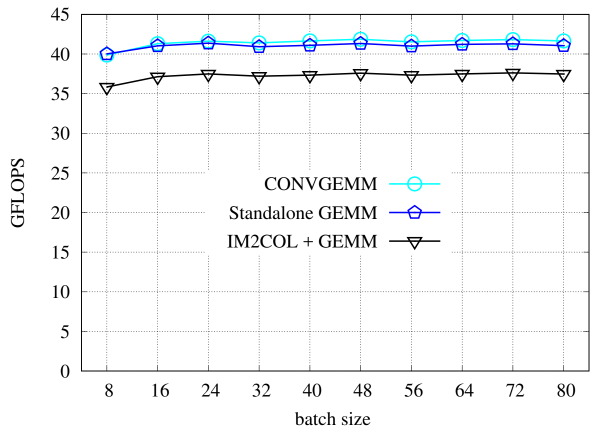


(f) GFLOPS, VGG16, 1 core

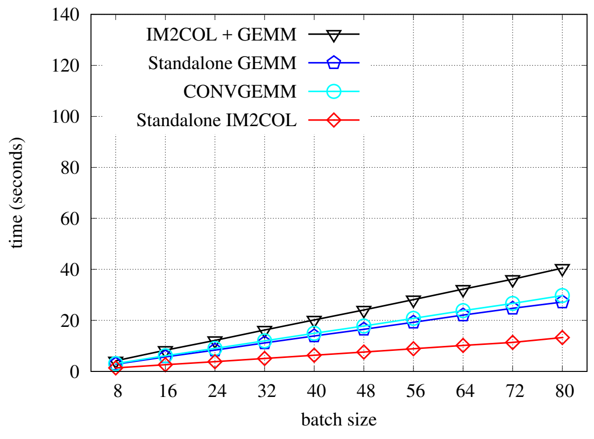
Figure 7: Execution time (left column) and performance (right column) obtained by the indirect convolution algorithms for AlexNet (top row), ResNet50 (middle row) and VGG16 (bottom row) on a single ARM Cortex-A57 core.



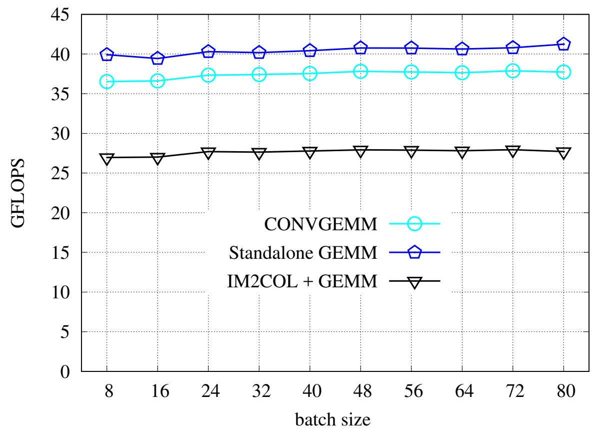
(a) Execution time, AlexNet, 4 cores



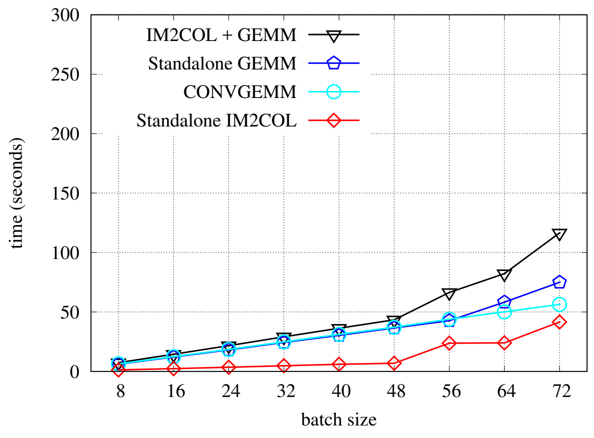
(b) GFLOPS, AlexNet, 4 cores



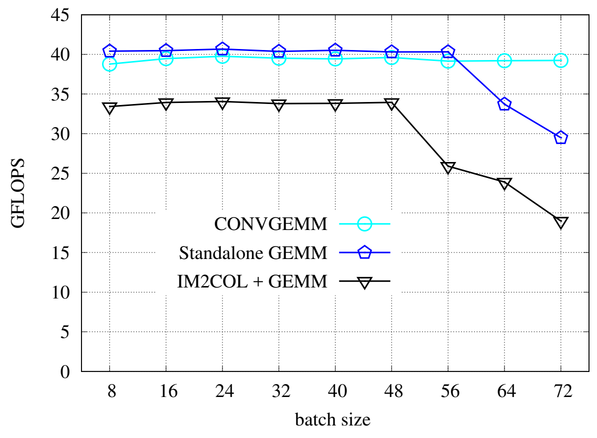
(c) Execution time, ResNet50, 4 cores



(d) GFLOPS, ResNet50, 4 cores



(e) Execution time, VGG16, 4 cores



(f) GFLOPS, VGG16, 4 cores

Figure 8: Execution time (left column) and performance (right column) obtained by the indirect convolution algorithms for AlexNet (top row), ResNet50 (middle row) and VGG16 (bottom row) using all four ARM Cortex-A57 cores.

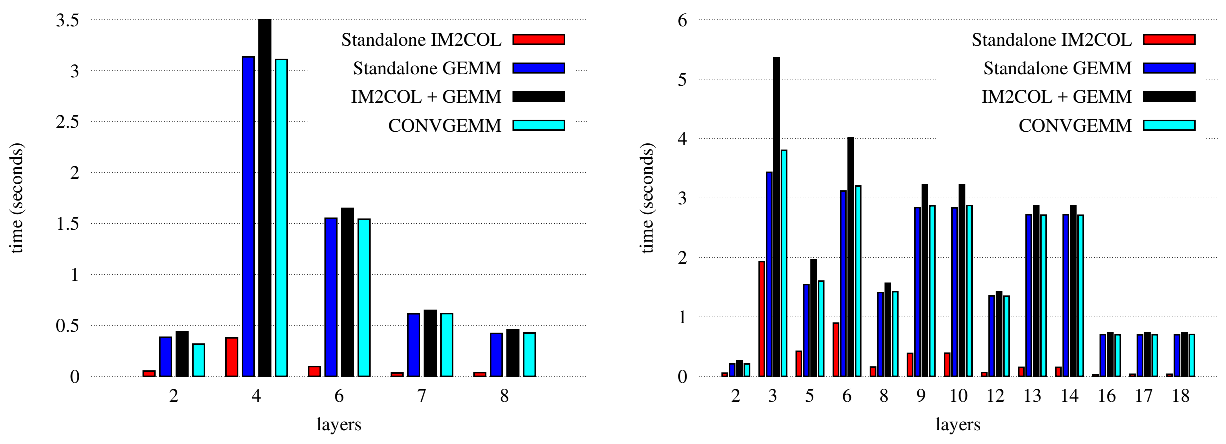


Figure 9: Execution time per layer obtained by the indirect convolution algorithms for AlexNet (left) and VGG16 (right) using all four ARM Cortex-A57 cores and a batch size $b = 32$.

There is a particular case worth of being discussed in some detail. Concretely, for the explicit IM2COL+GEMM approach, Figures 7f and 8f both show a notorious decrease in performance for the VGG16 when $b > 48$. This decline is caused by the large size of the intermediate matrices, which results in I/O swapping to disk. The negative effect in the performance is more notorious in the multicore experiment, as in this case the memory access patterns performed during the explicit IM2COL transform are more spread, increasing the effect of the 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.

To close the experimental analysis, Figure 9 reports the execution time to compute the convolutions required at each CNN layer in the AlexNet and VGG16 models. The plots there illustrate that the time required per layer significantly varies between different layers.

6 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 inference with large CNN models in memory bound systems. 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 other low-power architectures for which an optimized implementation of the BLIS micro-kernel exists (or can be developed).

The results in the experimental evaluation performed in this work show the remarkable performance advantage of the new CONVGEMM scheme on a representative low-power ARM-based multicore processor, which completely eliminates the workspace and performance overheads due to the utilization of an explicit IM2COL transform.

Acknowledgements

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*.

References

[1] 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 (DNNL). Available from `https://oneapi-src.github.io/oneDNN/`.

[2] Deep learning SDK documentation: cuDNN developer guide, 2020. Available from `https://docs.nvidia.com/deeplearning/sdk/cudnn-developer-guide/index.html`.

[3] Mobile AI Compute Engine documentation, 2020. Available from `https://mace.readthedocs.io/en/latest/`.

[4] NNPACK: Acceleration package for neural networks on multi-core CPUs, 2020. Available from `https://github.com/Maratyszcza/NNPACK`.

[5] Andrew Anderson et al. Low-memory GEMM-based convolution algorithms for deep neural networks. *CoRR*, abs/1709.03395, 2017.

[6] Tal Ben-Nun and Torsten Hoefler. Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. *ACM Computing Surveys*, 52(4):65:1–65:43, August 2019.

[7] Sandra Catalán, Francisco D. Igual, Rafael Mayo, Rafael Rodríguez-Sánchez, and Enrique S. Quintana-Ortí. Architecture-aware configuration and scheduling of matrix multiplication on asymmetric multicore processors. *Cluster Computing*, 19(3):1037–1051, 2016.

[8] Kumar Chellapilla, Sidd Puri, and Patrice 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-00112631`.

[9] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cuDNN: Efficient primitives for deep learning, 2014. arXiv preprint 1410.0759. Available from `https://arxiv.org/abs/1410.0759`.

[10] Minsik Cho and Daniel Brand. MEC: Memory-efficient convolution for deep neural network. In *Proceedings of 34th Int. Conference on Machine Learning – PMLR*, volume 70, pages 815–824, 2017.

[11] Li Deng, Jinyu Li, Jui-Ting Huang, Kaisheng Yao, Dong Yu, Frank Seide, Mike Seltzer, Geoff Zweig, Xiaodong He, Jason Williams, Yifan Gong, and Alex Acero. Recent advances in deep learning for speech research at Microsoft. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 8604–8608, May 2013.

[12] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain Duff. A set of level 3 basic linear algebra subprograms. *ACM Trans. on Mathematical Software*, 16(1):1–17, March 1990.

[13] Marat Dukhan. The indirect convolution algorithm. *CoRR*, abs/1907.02129, 2019. arXiv preprint 1907.02129. Available from `https://arxiv.org/abs/1907.02129`.

[14] Marat Dukhan, Yiming Wu, and Hao Lu. QNNPACK: open source library for optimized mobile deep learning, 2020. Available from `https://code.fb.com/ml-applications/qnnpack/`.

[15] Evangelos Georganas, Sasikanth Avancha, Kunal Banerjee, Dhiraj Kalamkar, Greg Henry, Hans Pabst, and Alexander Heinecke. Anatomy of high-performance deep learning convolutions on simd architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC 18, pages 66:1–66:12. IEEE Press, 2018.

[16] Source code repository. `https://gitlab.com/comtacts/convgemm`, 2020.

[17] Kazushige Goto and Robert van de Geijn. High performance implementation of the level-3 BLAS. *ACM Transactions on Mathematical Software*, 35(1):4:1–4:14, July 2008.

[18] Kazushige Goto and Robert A. van de Geijn. Anatomy of a high-performance matrix multiplication. *ACM Trans. on Mathematical Software*, 34(3):12:1–12:25, May 2008.

[19] Song Han, Huizi Mao, and William 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`.

[20] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[21] Greg Henry. BLAS based on block data structures. Theory Center Technical Report CTC92TR89, Advanced Computing Research Institute. Cornell University, 1992.

[22] NVIDIA Jetson TX2. `https://www.nvidia.com/es-es/autonomous-machines/embedded-systems/jetson-tx2/`, 2020.

[23] Jintao Ke, Hai Yang, Hongyu Zheng, Xiqun Chen, Yitian Jia, Pinghua Gong, and Jieping Ye. Hexagon-based convolutional neural network for supply-demand forecasting of ride-sourcing services. *IEEE Trans. on Intelligent Transportation Systems*, 20(11):4160–4173, 2019.

[24] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. ImageNet classification with deep convolutional neural networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, pages 1097–1105, USA, 2012. Curran Associates Inc.

[25] Andrew Lavin and Scott Gray. Fast algorithms for convolutional neural networks. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 4013–4021, 2016.

[26] Tze Meng Low, Francisco D. Igual, Tyler M. Smith, and Enrique S. Quintana-Ortí. Analytical modeling is enough for high-performance BLIS. *ACM Trans. on Mathematical Software*, 43(2):12:1–12:18, August 2016.

[27] Ningning Ma, Xiangyu Zhang, Hai-Tao Zheng, and Jian Sun. ShuffleNet V2: Practical guidelines for efficient CNN architecture design. In *Proceedings European Conference on Computer Vision - ECCV 2018. Lecture Notes in Computer Science*, volume 11218, pages 122–138, 2018.

[28] Maryam M. Najafabadi, Flavio Villanustre, Taghi M. Khoshgoftaar, Naeem Seliya, Randall Wald, and Edin Muharemagic. Deep learning applications and challenges in big data analytics. *Journal of Big Data*, 2(1):1, Feb 2015.

[29] OpenBLAS. `http://www.openblas.net`, 2015.

[30] OpenMP Architecture Review Board. OpenMP application program interface version 3.0, May 2008.

[31] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[32] Tyler M. Smith, Robert van de Geijn, Mikhail Smelyanskiy, Jeff R. Hammond, and Field G. Van Zee. Anatomy of high-performance many-threaded matrix multiplication. In *Proc. IEEE 28th Int. Parallel and Distributed Processing Symp.*, IPDPS'14, pages 1049–1059, 2014.

[33] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S. Emer. Efficient processing of deep neural networks: A tutorial and survey. *Proceedings of the IEEE*, 105(12):2295–2329, Dec 2017.

[34] Field G. Van Zee and Robert A. van de Geijn. BLIS: A framework for rapidly instantiating BLAS functionality. *ACM Trans. on Mathematical Software*, 41(3):14:1–14:33, 2015.

[35] R. Clint Whaley and Jack J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, SC 98, page 127, USA, 1998. IEEE Computer Society.

[36] Field G. Van Zee, Tyler M. Smith, Bryan Marker, Tze Meng Low, Robert A. Van De Geijn, Francisco D. Igual, Mikhail Smelyanskiy, Xianyi Zhang, Michael Kistler, Vernon Austel, John A. Gunnels, and Lee Killough. The BLIS framework: Experiments in portability. *ACM Trans. on Mathematical Software*, 42(2):12:1–12:19, June 2016.

[37] Jiajun Zhang and Chengqing Zong. Deep neural networks in machine translation: An overview. *IEEE Intelligent Systems*, 30(5):16–25, Sep. 2015.

[38] Jiyuan Zhang, Franz Franchetti, and Tze Meng Low. High performance zero-memory overhead direct convolutions. In *Proceedings of the 35th International Conference on Machine Learning – PMLR*, volume 80, 2018.

[39] Yulin Zhao, Donghui Wang, Leiou Wang, and Peng Liu. A faster algorithm for reducing the computational complexity of convolutional neural networks. *Algorithms*, 11(10):159, Oct 2018.