IEEE Access
Multidisciplinary : Rapid Review : Open Access Journal

Received 21 May 2023, accepted 29 May 2023, date of publication 6 June 2023, date of current version 14 June 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3283312

RESEARCH ARTICLE

Analysis and Optimization of Direct Convolution Execution on Multi-Core Processors

MIRCO MANNINO[1], BIAGIO PECCERILLO[1], ANDREA MONDELLI[2], AND SANDRO BARTOLINI[1]
[1]Department of Information Engineering and Mathematics, University of Siena, 53100 Siena, Italy
[2]Huawei Technologies Company Ltd., CB4 0WG Cambridge, U.K.

Corresponding author: Mirco Mannino (mannino@diism.unisi.it)

This work was supported in part by Huawei Technologies Research and Development (U.K.) Ltd., and in part by the Italian Ministry of University and Research.

ABSTRACT Nowadays, convolutional neural networks are among the most widely used types of deep learning networks thanks to their usefulness in many application domains. There are many efforts to find methods to increase their training and inference performance and efficiency. One of the most widely used technique to implement convolution consists of flattening tensors into 2D matrices and carrying out the operation through a matrix-matrix multiplication routine, which has highly optimized implementations in high-performance libraries. However, this kind of approach uses extra time and memory to transform and store the tensors involved. For this reason, *direct convolution* is becoming increasingly popular. Direct convolution can be implemented as a series of nested loops iterating over tensor dimensions and it does not require extra memory. In this work, we evaluate on various multi-core CPUs the performance and scalability effects deriving from different parallelization strategies, loop organizations, and SIMD-vectorization approaches with different compilers in relation with architectural aspects. We discuss each parameter thoroughly and distill our findings in a set of heuristics that can be used to quickly achieve a high-performance implementation in accordance to the underlying hardware and the characteristics of the convolutional layer at hand. By adopting a per-layer approach, we increase performance up to 60-70% compared to a static implementation for all the layers. Moreover, our results are comparable, or even better (up to 1.67× speedup) than matrix-matrix multiplication-based convolution in a multi-core system.

INDEX TERMS Convolutional neural networks, direct convolution, multi-core, multi-threading, performance evaluation.

I. INTRODUCTION AND MOTIVATION

Convolutional neural networks (CNNs) are widely used nowadays due to the large number of areas in which they can be applied, including computer vision [1], [2] (e.g., object recognition and object detection), bioinformatics [3], and natural language processing [4], [5].

State-of-the-art CNN architectures (e.g., AlexNet [6], VGG [7]) are generally composed of several layers. Input tensors are processed by the first layer of the convolutional part of the network. After that, the outcome is transformed by applying activation functions (e.g., ReLu) and pooling layers (e.g., MaxPooling) [8]. Several instances of these three types of layer are used to compose a CNN architecture. The aim of these layers is to extract more and more complex features from the input that will be used by the final part of the CNN to provide the network decision. Indeed, the final part of the architecture consists of the so-called fully-connected layer [8]. Figure 1 shows the scheme of a general CNN architecture.

The most onerous operation, from both computation time and energy consumption perspectives, is the convolution – therefore, its optimization in such directions is crucial for meeting the ever-increasing market requirements in both training and inference activities. In a convolution, each output element is calculated as the accumulation of several scalar

The associate editor coordinating the review of this manuscript and approving it for publication was Vlad Diaconita.

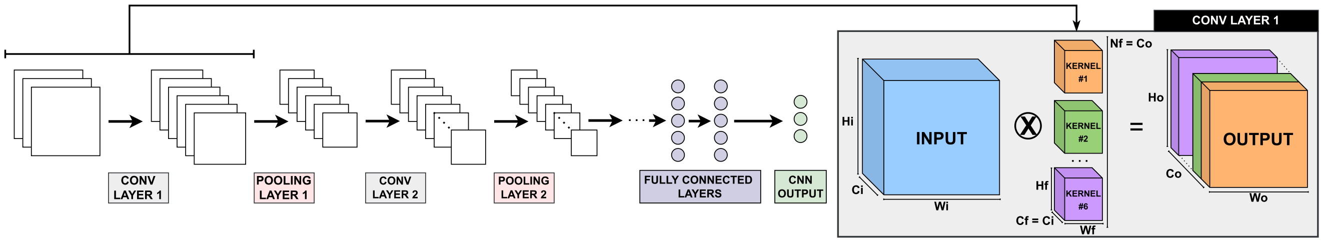


FIGURE 1. Scheme of a general CNN architecture (left). Convolutional layer example (right) represents "CONV LAYER 1."

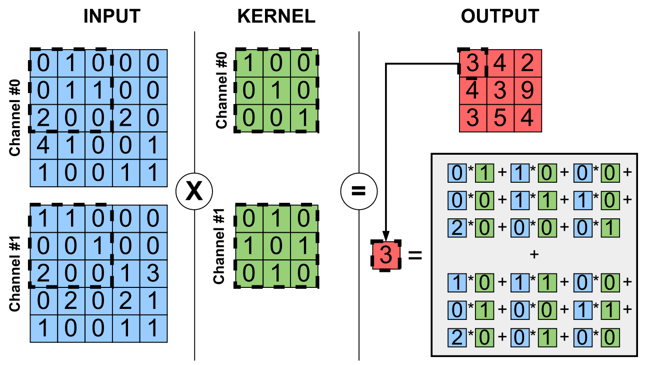


FIGURE 2. Example of convolution between a 5 × 5 × 2 input tensor and a 3 × 3 × 2 × 1 kernel tensor. The result is a 3 × 3 × 1 output tensor. The product of elements belonging to different channels of the input and kernel tensors are accumulated into a single output channel.

multiplications between elements read from input and kernel tensor. *Multiply-and-accumulate* is the main operation performed. Figure 2 shows a convolution operation example.

One of the most widely used methods to address convolution consists in the transformation of the involved tensors into two-dimensional matrices (*im2col* operation). Then, the computation is carried out by a matrix-matrix multiplication between the resulting matrices (*gemm* operation). Finally, the tensors are converted back to their original representation. This method is widely used since highly optimized matrix-matrix multiplication routines are available (e.g., OpenBLAS [9], Intel MKL [10]). However, even if the computation can be accelerated, the main drawbacks of this approach are the time needed to transform the tensors (*packing time*) and additional memory needed to store the transformed representations. Throughout the paper we use the term *im2col+gemm* to refer to this method based on matrix-matrix multiplications.

An alternative method that has been gaining increasing attention in recent times is the so-called *direct convolution*. It does not apply any transformations to the tensors and can be implemented through a series of nested loops that wrap an accumulation operation over the elements of the output tensor.

Direct convolution may seem straightforward to implement on CPU in a high-level language (e.g., C++). A working implementation can be obtained by simply translating the summations that make up the definition into six nested loops. However, this approach does not necessarily lead to the best performance achievable on a given system, as low computational and/or memory bandwidth may still be limiting factors. In order to reach high-performance on different architectures, *flexibility* should be taken into account: direct convolution should be conceived as a *parametric* operation, with parameters affecting the execution *order* of some groups of operations and their distribution/scheduling on the available computational resources. This may lead to different performance according to the parameters' values, despite the computation calculates the same result. Therefore, on a given system, design space exploration could be done in order to identify the set of parameters that allow achieving the best performance.

We identify four parameters that can be varied in such spirit. The first is the order in which loops are organized (i.e., loop orders and loop tiling). Different loop organizations induce different memory access patterns, and different ways of exploiting data locality, which have a well-known impact on performance. The second is the way the work is partitioned and assigned to parallel threads of execution. Since every output element is computed independently of the others, several levels of parallelism can be exploited to gain performance. Prior works [11], [12], [13], [14] confirm the importance of this aspect. Then, we analyze the role of SIMD-vectorization. At the heart of a convolution operation, we have floating-point multiply-accumulate instructions that can be grouped in SIMD instructions for increased throughput. Lastly, the compiler choice. Modern compilers have the ability to apply a long list of optimizations to high-level source code, such as instruction reordering, loop unrolling, and auto-vectorization, and they can make a significant difference in the execution.

However, these parameters are not independent of each other: for instance, the parallel execution effectiveness is affected by the concurrent usage of the last-level-cache and, therefore, by the working-set size of inner loops (thus, by their order). Or, as another example, compiler optimizations highly depend on the compiler ability to recognize opportunities based on code patterns, while different orders imply different code-structures. Furthermore, different machines may require different optimal parameter sets. Consequently, the choice of an optimal parameters' set is a challenging problem, as well as the definition of general criteria for selecting candidate values based on architectural features of the target machine.

In this work, we explore various combinations of values for the parameters introduced above on different architectures.

Our goal is to determine experimentally which combinations lead to the best performance in relation to the underlying hardware. We discuss the obtained performance and generalize our findings to help implementors to design direct convolution on their hardware. In summary, we use and combine the following elements:

- Different loop organizations (i.e., loop orders and loop tiling), with each being characterized by the type of data reuse and memory access pattern;
- Different parallelization strategies, in which we explore different ways of assigning output elements to threads of execution;
- Different SIMD-vectorization strategies, in which we group floating-point operations in one or more consecutive SIMD instructions;
- Different compilers to study their impact in terms of automatic optimizations, e.g., auto-vectorization;

The study presented in this work may change the way convolution is addressed in CNNs today. We show that an *im2col+gemm* implementation, despite being the most widespread solution, is not necessarily the best choice in terms of performance, as well as memory footprint. We conduct our analysis on multi-core CPUs. Where possible, nowadays the trend is to use GPUs, FPGAs, NPUs, and other ad-hoc accelerators for seeking higher performance/-efficiency than CPUs [15]. GPUs' massively parallel hardware has been successfully employed in the *im2col+gemm* convolution implementation, but also in direct convolution [16], [17] recently. State-of-the-art convolutional accelerators (e.g., [18], [19]) use specific dataflow structures that can be seen as *portions of direct convolution algorithm mapped in hardware*, since tensors are processed *spatially* without applying any transformations. Although these architectures differ substantially from multi-core CPUs, some of the results presented in this paper can be applied, generalized, and extended to write efficient direct convolution on GPUs or design ad-hoc accelerators. A full treatise covering also the application of direct convolution to these architectures is beyond the scope of this paper, and we will investigate it as future work.

The main contributions of this paper can be summarized as follows:

- We present and discuss an implementation of direct convolution based on four parameters (loop organization, parallelization strategy, SIMD vectorization, and compiler choice) that can be varied in search of peak performance;
- For each parameter, we do a thorough analysis of the possible values, its impact on performance and read the results in the light of architectural considerations;
- We explore different combinations of such parameters to identify the best performing values for both full network and per layer;
- We collect our *lessons learned* to help convolution designers to select the best parameters in accordance to the underlying hardware and the convolutional layer at hand.

II. CNN BACKGROUND AND DIRECT CONVOLUTION

Before going into the details of loop organizations, parallelization strategies, and vectorization strategies adopted, this section provides an introduction to basic concepts of convolution, with particular attention to the implications of adopting an implementation based on direct convolution.

In a CNN architecture, the convolutional layer is usually the most demanding in terms of running time. It involves an input tensor ($H_i \times W_i \times C_i$), a kernel (or filter) tensor ($H_f \times W_f \times C_f \times N_f$), and it produces an output tensor ($H_o \times W_o \times C_o$). In each convolutional layer, the number of kernels is the same as the output tensor's depth (i.e., number of channels) ($N_f = C_o$), while each kernel's depth is equal to the input tensor's depth ($C_f = C_i$).

Output elements are calculated as follows:

$$\mathbf{O}_{h_o,w_o,c_o} = \sum_{n=0}^{H_f}\sum_{m=0}^{W_f}\sum_{i=0}^{C_i} \mathbf{I}_{h_o\cdot s+n,w_o\cdot s+m,i} \cdot \mathbf{K}_{n,m,i,c_o} \tag{1}$$

where **O**, **I**, and **K** are output, input, and kernel tensors, respectively, and s represents the *stride*. The latter is a convolution parameter that defines the number of steps taken by the convolution filter in both directions as it slides over the input tensor. Another convolution parameter is the so-called *padding*. It represents the number of extra rows and columns added to the input tensor's margins to be able to operate seamlessly on its border. These two parameters are used to control output tensor spatial size (i.e., height and width):

$$\begin{aligned} H_o &= \frac{(H_i - H_f + 2\cdot padding)}{stride} + 1 \\ W_o &= \frac{(W_i - W_f + 2\cdot padding)}{stride} + 1 \end{aligned} \tag{2}$$

Equation 1 shows that input height and width indexes are not used as *main indexes* of summations, but they are obtained by combining the indexes of output and kernel tensors. Figure 2 shows an example of a convolution operation between a $5 \times 5 \times 2$ input tensor and a $3 \times 3 \times 2 \times 1$ kernel tensor, producing a $3 \times 3 \times 1$ output tensor.

Direct convolution, that in its naive form can be implemented as a series of nested loops, is a straight implementation of Equation 1. Its characterization depends on how the loops are arranged among themselves. Notably, the order of loops has a direct impact on memory access patterns and utilization of computational resources. In Section III-A, we discuss these aspects for the selected loop orders.

Moreover, each element of the output tensor can be computed independently of the others, making direct convolution highly parallelizable. There are two levels of parallelism that can be exploited to speedup the performance of direct convolution. The first is a coarse-grained parallelism that can be achieved through multi-threading, exploiting multiple cores of the CPU. The second is related to a finer-grain spatial parallelism, obtained through SIMD instructions, that

TABLE 1. Notation of shape parameters.

Tensor type	Height	Width	N. of channels	N. of items
Input	H_i	W_i	C_i	
Kernel	H_f	W_f	C_f	N_f
Output	H_o	W_o	C_o	

```
void loop_order_N1(const DType I, const DType K, DType O)
{
  for(int l = 0; l < Ho; l++)
    for(int n = 0; n < Hf; n++)
      for(int m = 0; m < Wf; m++)
        for(int i = 0; i < Ci; i++)
          for(int k = 0; k < Wo; k++)
            for(int j = 0; j < Co; j++)
              O[l][k][j] +=
                K[n][m][i][j] *
                I[n*stride+l][m*stride+k][i];
}
```

Listing. 1. Loop order N.1.

```
void loop_order_N2(const DType I, const DType K, DType O)
{
  for(int n = 0; n < Hf; n++)
    for(int m = 0; m < Wf; m++)
      for(int i = 0; i < Ci; i++)
        for(int l = 0; l < Ho; l++)
          for(int k = 0; k < Wo; k++)
            for(int j = 0; j < Co; j++)
              O[l][k][j] +=
                K[n][m][i][j] *
                I[n*stride+l][m*stride+k][i];
}
```

Listing. 2. Loop order N.2.

allow carrying out a specific operation (e.g., addition and multiplication) over several adjacent scalar values at once (e.g., 8 single-precision scalar values can be simultaneously managed using 256-bit SIMD instructions). In direct convolution, outermost loops are parallelized using a multi-threaded approach, while innermost loops are spatially parallelized using SIMD operations (Section III-C). The parallelization strategy is a key contribution in the direct convolution characterization. Section III-B shows the strategies used in this work and their direct implication on their memory use.

III. DIRECT CONVOLUTION PARAMETERS

We identify four parameters to model direct convolution: loop organization (i.e., loop orders and loop tiling), parallelization strategy, SIMD vectorization, and compiler choice. This section describes the main characteristics of each parameter. In particular, Section III-A and Section III-B explain the loop orders and the parallelization strategies, respectively, used to implement the different versions of direct convolution. Section III-C shows how the other two parameters, i.e, SIMD vectorization and compiler choice, are related to direct convolution and how they are used in this work.

A. LOOP ORDERS AND LOOP TILING

As mentioned in previous sections, the order in which loops of direct convolution are organized does not affect result correctness. However, it changes the memory access pattern and the usage of architectural resources.

Before presenting the orders of the loops used, it is important to know how tensors are arranged in memory. Input and output tensors are three-dimensional tensors, for which we use the HWC memory layout (i.e., most local dimensions in

```
void loop_order_N3(const DType I, const DType K, DType O)
{
  for(int l = 0; l < Ho; l++)
    for(int n = 0; n < Hf; n++)
      for(int m = 0; m < Wf; m++)
        for(int k = 0; k < Wo; k++)
          for(int i = 0; i < Ci; i++)
            for(int j = 0; j < Co; j++)
              O[l][k][j] +=
                K[n][m][i][j] *
                I[n*stride+l][m*stride+k][i];
}
```

Listing. 3. Loop order N.3.

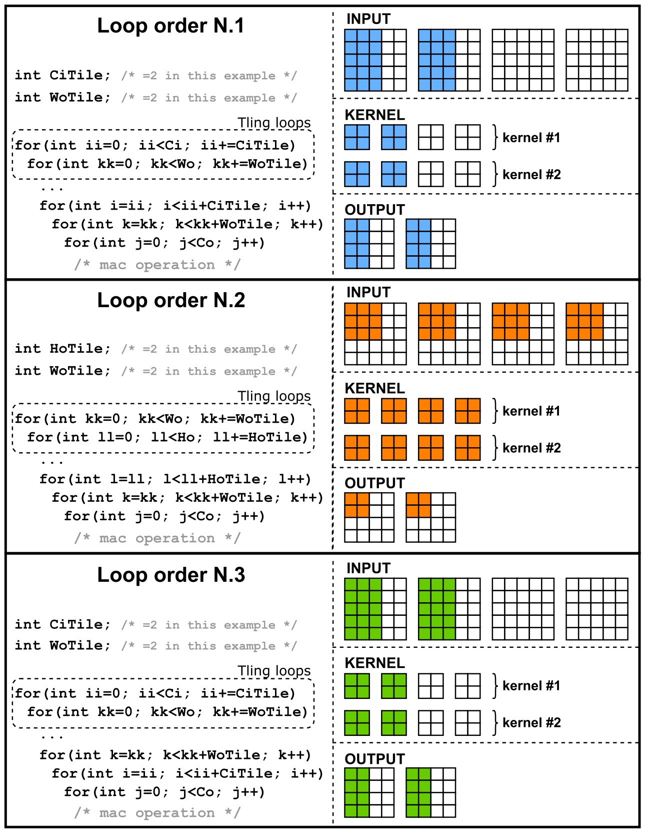
order are depth, width, and height), while the kernel tensor has four dimensions and it is arranged in memory according to the HWCN layout (i.e., most local dimensions in order are number of kernels, depth, width and height). Table 1 shows the notations used for the shape parameter for each type of tensor. Since the output tensor is arranged with depth (C_o) as the most contiguous dimension in memory, it is typically convenient to iterate through that dimension in the innermost loop. This choice fosters data locality and easier handling of SIMD instructions. The order of the outermost loops is selected according to observations resulting from the use of SIMD units and tensors' access patterns. Three different loop orders are analyzed, according to the following observations:

Loop order N.1 This loop order allows accessing to a different contiguous output element during each consecutive $W_o * C_o$ iterations. According to [11], this helps not to stall SIMD units, since every output element accessed in one iteration does not depend on the previous output elements. For this reason, iteration over output depth and width are in the innermost loops. The next three loops are used to iterate over the dimensions of each kernel. The outermost loop is dedicated to output height iteration (see Listing 1).

Loop order N.2 This loop order has the three innermost loops that iterate over all the output dimensions. As in the order N.1, the three innermost loops guarantee an output memory access in which every element is different from previous iterations. What characterizes this order of loops is the number of different consecutive accessed elements: $W_o * C_o$ and $H_o * W_o * C_o$ in order N.1 and N.2, respectively. The next three loops are used, as in order N.1, for the iteration of each kernel (see Listing 2).

Loop order N.3 This loop order is slightly different from the previous two, since the two innermost loops are dedicated to the iteration of output and kernel depths. By doing so, stalling of SIMD units is more probable but there is higher reuse of kernel data, supporting better utilization of cache memories. The next loop is used to iterate over output width; then height and width of kernel are iterated and, finally, the outermost loop is dedicated to output height iteration (see Listing 3).

Accessing the same output element in two consecutive iterations produces a *Read-After-Write* dependency that could be harmful when SIMD units are used [11]. SIMD units have a higher throughput than scalar units, at the cost of a higher latency of individual instructions. On the other hand, accessing different elements in every consecutive iteration



Loop order N.1

```
int CiTile; /* =2 in this example */
int WoTile; /* =2 in this example */
for(int ii=0; ii<Ci; ii+=CiTile)
  for(int kk=0; kk<Wo; kk+=WoTile)
    ...
    for(int i=ii; i<ii+CiTile; i++)
      for(int k=kk; k<kk+WoTile; k++)
        for(int j=0; j<Co; j++)
          /* mac operation */
```

Loop order N.2

```
int HoTile; /* =2 in this example */
int WoTile; /* =2 in this example */
for(int kk=0; kk<Wo; kk+=WoTile)
  for(int ll=0; ll<Ho; ll+=HoTile)
    ...
    for(int l=ll; l<ll+HoTile; l++)
      for(int k=kk; k<kk+WoTile; k++)
        for(int j=0; j<Co; j++)
          /* mac operation */
```

Loop order N.3

```
int CiTile; /* =2 in this example */
int WoTile; /* =2 in this example */
for(int ii=0; ii<Ci; ii+=CiTile)
  for(int kk=0; kk<Wo; kk+=WoTile)
    ...
    for(int k=kk; k<kk+WoTile; k++)
      for(int i=ii; i<ii+CiTile; i++)
        for(int j=0; j<Co; j++)
          /* mac operation */
```

FIGURE 3. Loop tiling applied to loop orders N.1, N.2, and N.3. On the left, we show the newly-added tiling loops and the modifications of the three innermost loops, for each order considered. On the right, we show the elements of input, kernel, and output tensors accessed during the first tile computation. In the example, tile size is set to 2 in all the cases.

could lead to the loss of data locality, that does not allow a proper exploitation of cache memories. With loop orders N.1 and N.2, the same output element is accessed every $C_o * W_o$ iterations, while with loop order N.3, it happens every C_o iterations. Loop order N.3 accesses output elements giving more importance to the cache locality, since its innermost iterations are along the most contiguous dimensions of output and kernel tensors. On the other hand, with loop orders N.1 and N.2, there is a longer reuse distance of the same output elements, which makes SIMD unit stall less likely.

1) LOOP TILING

Loop tiling is a well-known optimization technique used to increase data locality in a nested loop computation. It consists of dividing the workload in smaller blocks (i.e., tiles), aiming to increase data locality and performance. Loop tiling implementation requires additional loops that iterate over, and within, each tile. Loop orders N.1, N.2, and N.3 can be further optimized using loop tiling. There are 6 dimensions that can be used to divide the workload into smaller blocks (i.e., H_o, W_o, C_o, H_f, W_f, and C_i). The choice of the dimension (or *dimensions*) to select to apply loop tiling is strongly related to the order of the loops. Each of the proposed orders needs a targeted tiling strategy.

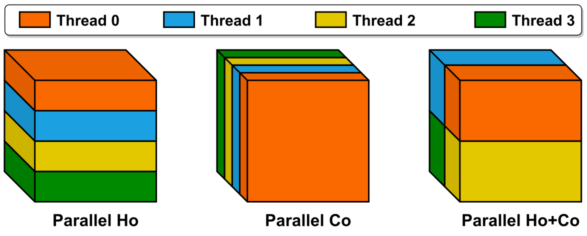


FIGURE 4. Output tensor regions assigned to 4 threads using parallel-Ho, parallel-Co and parallel-Ho+Co.

We apply loop tiling considering the innermost loops of each order. Apply tiling along the dimensions iterated in the innermost loop leads to an expensive overhead and, thus, a performance decay. Loop orders N.1, N.2, and N.3 all have the innermost loop that iterates over the depth of the output (C_o), therefore, this dimension is not used for partitioning the workload. Having excluded the output depth dimension, for each order we used the dimension iterated in the second innermost loop. By using only one dimension to apply tiling, it can happen that the workload is not reduced sufficiently to achieve a performance increase. For this reason, we also use the third innermost loop to apply tiling. Thus, loop orders N.1 and N.3 use tiling along output width (W_o) and input depth (C_i), while tiling in loop order N.2 is applied along output width (W_o) and height (H_o).

Figure 3 shows how loop orders change after tiling (on the right) and which elements of each tensor are accessed by a single tile (on the left). Loop orders N.1 and N.3 access the same elements in a tile, but they differ in the order in which elements are accessed.

Applying tiling along the input depth (loop orders N.1 and N.3) allows reducing the number of elements accessed by both input and kernel in a single tile. Consequently, a single output element needs to be processed by more than one tile in order to be completed. On the contrary, apply tiling only on output dimensions (loop order N.2) allows carrying out the whole computation of an output element in a single tile, avoiding accessing it again in a different tile.

B. PARALLELIZATION STRATEGY

Since every output element can be computed independently of the others, several parallelization strategies can be used. Changing the way a group of output elements is assigned to each thread reflects on architectural resource utilization and consequently on performance.

For the following discussion, we assume that T indicates the number of available threads. We use three types of parallelization strategies:

Parallel-Ho This parallelization strategy assigns to each thread a portion of output rows. Each thread is in charge to compute the results of H_o/T output rows, while kernel and input tensors are accessed sequentially. In particular, accessed input tensor region is reduced by a factor that depends on T and the height of each kernel (H_f), since each output row computation requires H_f lines of the input. Accessed kernel tensor memory is not reduced at

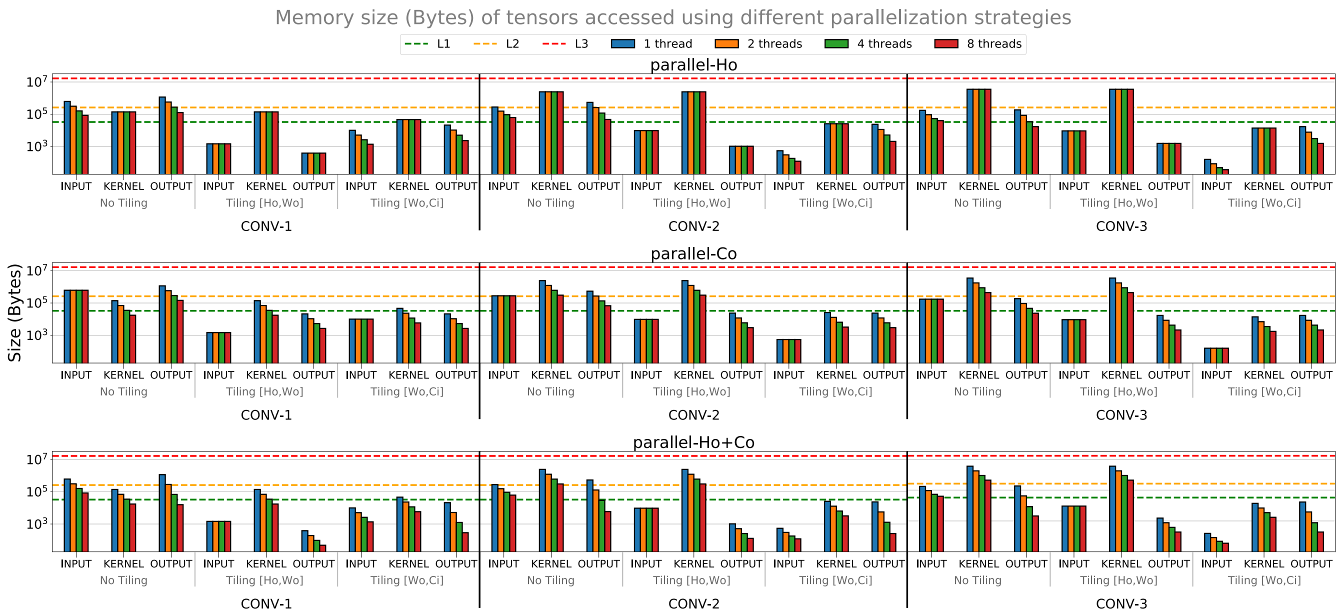


FIGURE 5. **Memory size (bytes) of input, kernel, and output tensors accessed per thread in each parallelization strategy for 1, 2, 4, and 8 threads, using the first three convolutional layers of AlexNet, labeled as CONV-1, CONV-2, and CONV-3. The other layers are not shown because their memory size is similar to that of CONV-3. The figure shows also the memory size of tensors when the tiling is applied with the smallest tile sizes for each tiling strategy considered (i.e., $H_{o\_tile} = W_{o\_tile} = 1$ and $W_{o\_tile} = C_{i\_tile} = 1$). Dashed lines represent the cache sizes of an Intel Core i9-9900K CPU (i.e., L1 = 32KB, L2 = 256KB, and L3 = 16MB).**

all, because each output channel needs a whole kernel to be computed. Thus, each thread has to use the whole kernel tensor.

Parallel-Co This parallelization strategy aims to reduce the amount of kernel tensor memory accessed per thread. In fact, in this case, the output tensor is partitioned along the depth dimension, resulting in an assignment of a portion of the channels to each of the available threads. Thus, each thread computes C_o/T output channels, corresponding to a total of $H_o \times W_o \times (C_o/T)$ output elements. Input memory accesses do not decrease as the number of threads increases, since the full input tensor is required to compute a single channel of the output.

Parallel-Ho+Co Previous parallelization strategies aim to reduce the output elements to be computed per thread by slicing along specific dimension of it (height or depth). This parallelization strategy, conversely, targets a combination of both. In this case, output tensor elements are assigned to each thread by splitting the output along both height and depth.

Figure 4 shows the output tensor portion assigned to each thread using parallel-Ho, parallel-Co, and parallel-Ho+Co.

Additional parallelization strategies can be adopted leveraging the output width (W_o). However, since it is a quite common situation to have convolutional layers with the same output height and width ($H_o = W_o$) [6], [7], [20], using W_o to parallelize leads to thread workloads similar to those obtained with parallel-Ho, but with lower data-locality. Thus, parallelization strategies involving the W_o are not taken into account.

Using parallel-Ho+Co, the output elements can be partitioned along two different dimensions. We choose to always split the depth (C_o) dimension into two partitions, while the height (H_o) dimension is split according to the number of threads. For instance, in case of 8 available threads, output rows are divided into four portions and each thread computes $C_o/2$ channels of a portion of rows. Since output depth (C_o) is the most contiguous dimension in memory, splitting it into only two partitions, and split height (H_o) dimension according to the number of threads, allows maintaining data locality as the number of threads increases. This is because a higher number of threads results in more partitions along the less contiguous dimension (output height H_o) and not along the more contiguous one (output depth C_o).

The amount of memory accessed by each thread depends on parallelization and tiling strategies. Figure 5 shows the amount of accessed memory (bytes) per thread, for each type of tensor, using up to 8 threads and single-precision values, in the first three convolutional layers of AlexNet (i.e., CONV-1, CONV-2, CONV-3), in three different scenarios: 1) no tiling; 2) tiling applied along W_o and C_i, considering the smallest tile size (i.e., $W_{o\_tile} = C_{i\_tile} = 1$); 3) tiling applied along H_o and W_o, considering the smallest tile size (i.e., $H_{o\_tile} = W_{o\_tile} = 1$). The dashed lines in Figure 5 show the cache sizes of an Intel Core i9-9900K CPU (i.e., L1 = 32KB, L2 = 256KB, and L3 = 16MB). The figure highlights that loop execution order, tiling and parallelization can induce different working set sizes per thread, and thus opportunities to exploit faster average memory access time due to reduced usage of L2 and L3 caches.

There are different situations, in terms of reduction of memory accessed per thread, depending on the parallelization strategy. Parallel-Ho strategy reduces the amount of memory accessed of input and output tensors, while parallel-Co reduces the total memory accessed of output and kernel tensors. Parallel-Ho+Co, being a hybrid strategy between the other two, allows reducing the amount of accessed memory of every tensor. However, when loop tiling is not used (*No Tiling* in Figure 5) the total workload is above, or at most slightly below, the L2 cache threshold. Using tiling (*Tiling*$[H_o, W_o]$ and *Tiling* $[W_o, C_i]$ in Figure 5) allows reducing the working set under the L2 threshold and, thus, achieve higher performance. *Tiling*$[H_o, W_o]$ decreases, on average, the memory accessed of 66%, 63%, and 60% for CONV-1, CONV-2, and CONV-3, respectively. Instead, *Tiling*$[W_o, C_i]$ allows higher memory reduction with a decrease of accessed memory, on average, of 87%, 95%, and 95% for CONV-1, CONV-2, and CONV-3, respectively. This is an expected situation since, apart from initial layers, input depth (C_i) has higher values compared to output spatial dimensions (H_o and W_o), leading to smaller tile sizes when the former is used for loop tiling.

C. SIMD-VECTORIZATION

Modern architectures are equipped with SIMD units that allow applying the same operation (e.g., addition) to multiple data elements simultaneously. SIMD unit registers, in most common architectures, can handle 128-, 256-, and 512-bit wide data. The number of scalar elements computed by a SIMD instruction depends on SIMD register width and data type used (e.g., 256-bit wide registers can operate either on 8 32-bit single-precision or 4 64-bit double-precision scalar elements). They are very suitable for convolution and can provide a significant performance increase. In particular, the main operation involved in the computation is a *multiply-and-accumulate*, that can be addressed through dedicated *FMA units* present in the architecture. There are three main ways to use SIMD instructions within an application [21]:

Assembly instructions Explicit assembly code of SIMD instructions is inserted in the source code. This method implies more effort for the programmer and a need for more attention to memory management.

Intrinsic functions Compiler-provided type of functions that are replaced with a sequence of low-level instructions at compile-time. They are easier to use than assembly code, since there is a higher level of abstraction.

Compiler auto-vectorization By enabling optimization flags and using specific data access patterns, compiler can insert SIMD instructions during the compilation phase.

Using SIMD instructions, the throughput can increase significantly, at the cost of higher latency of individual instructions. In computations involving multiple reads and writes of the same addresses, e.g., the accumulation operation over the output elements in direct convolution, it is important that the access to a memory location occurs at least after a certain number of cycles (i.e., SIMD unit latency cycles) to avoid stalling SIMD units and hide their latency [11].

TABLE 2. Convolutional layers selected from AlexNet, ResNet and VGG. "CNN" column indicates the network from which the layer was extracted. Input tuples correspond to (H_i, W_i, C_i) and kernel tuples correspond to (H_f, W_f, C_f, N_f). Layers are divided into three groups, according to their position along the CNN architecture.

	Layer ID	CNN	Input	Kernel	Stride
initial	0	AlexNet	(227, 227, 3)	(11, 11, 3, 96)	4
	1	ResNet	(230, 230, 3)	(7, 7, 3, 64)	2
	2	VGG	(226, 226, 3)	(3, 3, 3, 64)	1
intermediate	3	AlexNet	(31, 31, 96)	(5, 5, 96, 256)	1
	4	ResNet	(58, 58, 64)	(3, 3, 64, 64)	1
	5	ResNet	(58, 58, 64)	(3, 3, 64, 128)	2
	6	VGG	(58, 58, 128)	(3, 3, 128, 256)	1
	7	ResNet	(30, 30, 128)	(3, 3, 128, 128)	1
	8	VGG	(30, 30, 256)	(3, 3, 256, 512)	1
final	9	VGG	(16, 16, 512)	(3, 3, 512, 512)	1
	10	AlexNet	(15, 15, 384)	(3, 3, 384, 256)	1
	11	ResNet	(9, 9, 512)	(3, 3, 512, 512)	1

Direct convolution is very suitable to use SIMD instructions: in its optimized versions [11], [12], [13], [14], there is heavy use of them.

Another widely used technique to increase *instruction level parallelism* (ILP) and, consequently, performance is the so-called *loop-unrolling*. It can be implemented by replicating, in the loop body, the instructions for the loop to occur multiple times and changing the counter variables accordingly. It aims to promote instruction level parallelism opportunity, minimizing the total number of instructions executed by reducing loop conditional instructions and the number of updates of the loop counter variable. Also in this case, it can be used in different ways: 1) manually unroll loops in source code or 2) let the compiler unroll loops at compile time. Although it can increase performance, excessively relying on it could be counter-productive due to an increase in code size. In general, in modern architectures, loop unrolling can be safely delegated to the compiler [22].

We adopt both an explicit approach in which we insert SIMD instructions in the source code, through intrinsic functions and loop unrolling, and an indirect one in which we rely on compiler's auto-vectorization capabilities. In the first case, we use 256-bit wide registers, which can handle 8 single-precision values at a time. Through loop unrolling, we group 8 operations in 1 SIMD instruction (*simd-8*), 16 in 2 SIMD instructions (*simd-16*), and 32 in 4 SIMD instructions (*simd-32*). In the second case, we adopt a version in which the compiler can exploit SIMD instructions freely (*simd-auto*). Since the compiler choice plays a crucial role from a performance point of view, we include it among the parameters to be evaluated for the implementation of direct convolution. We analyse and discuss the performance obtained by using three of the most widely used compilers: GCC, LLVM, and Intel compiler.

IV. DISCUSSION AND EXPERIMENTAL RESULTS

A. EXPERIMENTAL SETUP

We run our experiments on Intel Core i9-9900K (Coffee Lake) and AMD Ryzen 9 5900X (Zen 3) architectures. Table 3 summarizes their main characteristics. In the follow-

TABLE 3. Architectures main characteristics.

	Intel Core i9-9900K	AMD Ryzen 9 5900X
N. physical cores	8	12
Clock frequency	3.6 GHz	3.7 GHz
L1D cache	32 KB	32 KB
L2 cache	256 KB	512 KB
L3 cache (shared)	16 MB	64 MB

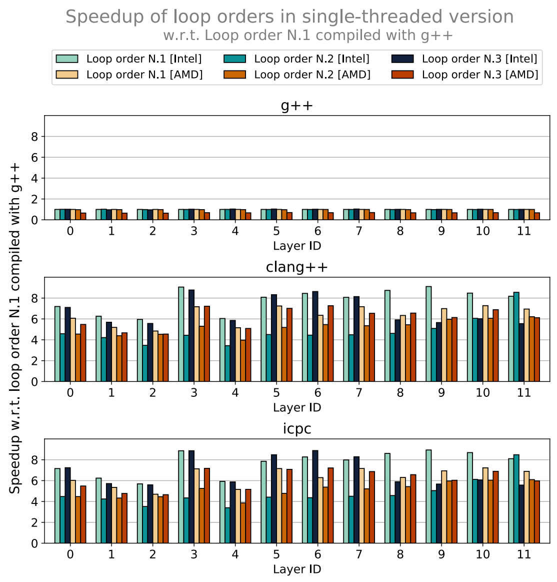


FIGURE 6. Speedup, w.r.t. g++ version of loop order N.1, of selected convolutional layers, obtained using loop orders N.1., N.2 and N.3 and compiling with g++, clang++, and icpc.

ing experiments, we refer to these systems as Intel CPU and AMD CPU, respectively.

We implement direct convolution in C++, using single-precision floating-point values (4 bytes). We conduct our experiments against convolutional layers of AlexNet [6], ResNet [20], and VGG [7]. We show a per-layer analysis and a full-network comparison against *im2col+gemm* implementations. For the former, we select a subset of representative layers from each of the CNN architectures. Table 2 shows the selected layers, divided into three groups according to their position along the CNN architecture: initial layers (0,2), intermediate layers (3-8), and final layers (9-11). We explore different loop orderings, loop tiling approaches, parallelization strategies, and compile our code with three different compilers: g++ version 12.1.0, clang++ version 15.0.6, and Intel compiler (icpc) version 2022.1.0. This way, we analyze the optimization strategies put in place by each of them.

For the *im2col+gemm* implementations we implement it by applying the *im2col* function from the popular deep learning framework Caffe [23] for tensor transformation, and we use expert-provided matrix-matrix multiplication routines from openBLAS version 0.3.21 [9] and Intel MKL version 2023.0 [10], which take advantage of various optimization techniques, such as loop unrolling and SIMD vectorization. In each analysis, the execution time results are obtained taking the median value of 1000 runs.

B. LOOP ORDERS

First, we analyze loop orderings, discussed in Section III-A, in a single-threaded version, using all our compilers. In this case, we compile with *-O3* optimization flag. Figure 6 shows the speedup with respect to g++ version of order N.1, for all loop orders and all compilers, using both Intel and AMD architectures.

Using clang++ and icpc, obtained results are quite similar on both architectures. In fact, loop order N.1 is always the best among all the orders. Although order N.1 has the best performance, the performance of the other two orders depends on the type of convolutional layer. Order N.3 behaves similar to order N.1 in the initial layers, characterized by a small number of input channels. Since the second innermost loop of order N.3 is the one that iterates over input channels, when their number is low, order N.1 and N.3 have similar performance. Order N.3 seems to suffer from a high number of input channels. Loop order N.2 shows up to 50% performance degradation, compared to the others, in the first convolutional layers, which are characterized by a larger output tensor size. Performance degradation of order N.2 is higher (i.e., about 80-100%) in the intermediate layers, characterized by tensors with medium spatial size and medium/high depth values. In the final layers, the three orders achieve similar performance.

The situation is slightly different using g++. In particular, on Intel CPU the performance is about the same for all the tested loop orders. On AMD CPUs, only loop order N.1 and loop order N.2 have similar performance, while order N.3 shows a performance decay in all the layers (about 30-50% slowdown).

Figure 6 shows that clang++ and icpc versions have a speedup at least 4× compared to g++. This result can be explained by looking at the assembly code generated by each compiler. SIMD instructions are present in assembly code generated by icpc and clang++, whereas there are none in the one generated by g++. On Intel CPU, g++ has a slower execution time for loop orders N.1, N.2, and N.3, with an average of 15%, 12% and 50% performance decrease compared to clang++ and icpc. On AMD CPU, the execution time of g++ versions are 43%, 29%, and 35% slower than clang++ and icpc versions for loop orders N.1, N.2, and N.3, respectively.

C. LOOP TILING

We analyze the effects of loop tiling on loop orders N.1, N.2, and N.3. Tiling is applied along output width (W_o) and input depth (C_i) for orders N.1 and N.3, while it is applied along output height (H_o) and width (W_o) for order N.2 (as discussed in Section III-A). For each loop order, we determine the tiling dimension (or dimensions) along which there is maximum speedup with respect to a non-tiled code. Table 4 shows, for each CPU and compiler pair, the best speedups and the tiling dimensions that provide such speedups.

Loop order N.2 is the one that benefits more from tiling, with speedups reaching up to 2× with respect to non-tiled

TABLE 4. Speedup of loop tiling applied to selected convolutional layers, w.r.t. non-tiled version, using loop orders N.1, N.2, and N.3, compiling with g++, clang++, and icpc. The table shows, for each combination *CPU-compiler*, the speedup achieved by adopting loop tiling and the dimensions on which such loop tiling is applied (*T.D.*, or "tiling dimensions" in the tables).

Layer	Loop order N.1		Loop order N.2		Loop order N.3	
ID	Speedup	T.D.	Speedup	T.D.	Speedup	T.D.
0	1.04	Ci	1.25	Ho+Wo	1.20	Ci
1	1.05	Wo+Ci	1.38	Ho	1.52	Ci
2	1.03	Ci	1.66	Ho	1.34	Wo+Ci
3	1.02	Ci	1.38	Ho+Wo	1.09	Wo
4	1.01	Ci	1.31	Ho	1.09	Wo
5	1.10	Ci	1.59	Ho	1.11	Wo+Ci
6	1.02	Wo	1.19	Ho	1.09	Wo
7	1.11	Wo	1.39	Wo	1.11	Ci
8	1.06	Ci	1.30	Ho	1.16	Wo
9	1.03	Ci	1.26	Wo	1.22	Wo
10	1.02	Ci	1.25	Ho	1.21	Wo
11	1.09	Wo	1.11	Ho	1.27	Ci

(a) Intel, g++

Layer	Loop order N.1		Loop order N.2		Loop order N.3	
ID	Speedup	T.D.	Speedup	T.D.	Speedup	T.D.
0	1.05	Ci	1.33	Ho	1.79	Ci
1	1.13	Ci	1.41	Ho	1.80	Ci
2	1.07	Wo	1.24	Ho	1.83	Ci
3	1.09	Ci	1.23	Ho	1.01	Ci
4	1.08	Wo+Ci	1.28	Ho	1.07	Wo
5	1.14	Wo	1.37	Ho	1.08	Ci
6	1.02	Ci	1.07	Ho	1.04	Ci
7	1.07	Ci	1.16	Ho	1.19	Ci
8	1.01	Ci	1.05	Ho	1.07	Ci
9	1.03	Ci	1.23	Wo	1.19	Ci
10	1.05	Wo	1.06	Ho	1.06	Wo
11	1.14	Ci	1.00	Ho	1.24	Ci

(b) AMD, g++

Layer	Loop order N.1		Loop order N.2		Loop order N.3	
ID	Speedup	T.D.	Speedup	T.D.	Speedup	T.D.
0	1.02	Wo	1.48	Ho	1.01	Wo
1	1.01	Wo	1.35	Ho	1.01	Wo
2	1.06	Wo	1.68	Ho	1.03	Ci
3	1.02	Wo+Ci	2.01	Ho	1.08	Ci
4	1.02	Wo	1.72	Ho	1.02	Ci
5	1.04	Wo+Ci	1.85	Ho	1.04	Ci
6	1.09	Wo+Ci	1.89	Ho	1.13	Ci
7	1.02	Wo	1.83	Ho	1.10	Ci
8	1.04	Wo+Ci	1.86	Ho	1.56	Ci
9	1.02	Ci	1.80	Ho	1.62	Ci
10	1.02	Ci	1.44	Ho	1.24	Ci
11	1.10	Ci	1.07	Wo	1.56	Ci

(c) Intel, clang++

Layer	Loop order N.1		Loop order N.2		Loop order N.3	
ID	Speedup	T.D.	Speedup	T.D.	Speedup	T.D.
0	1.01	Wo	1.36	Ho	1.00	Wo
1	1.02	Wo	1.22	Ho	1.01	Wo
2	1.12	Wo	1.10	Ho+Wo	1.03	Wo
3	1.06	Wo+Ci	1.35	Ho	1.05	Ci
4	1.03	Wo	1.36	Ho	1.00	Wo
5	1.05	Wo	1.36	Ho	1.02	Ci
6	1.21	Wo+Ci	1.29	Ho+Wo	1.07	Wo+Ci
7	1.04	Wo	1.31	Ho	1.05	Ci
8	1.13	Wo+Ci	1.32	Ho+Wo	1.13	Wo+Ci
9	1.01	Wo+Ci	1.22	Ho+Wo	1.21	Ci
10	1.01	Wo	1.17	Ho	1.08	Wo+Ci
11	1.03	Wo	1.10	Ho	1.21	Ci

(d) AMD, clang++

Layer	Loop order N.1		Loop order N.2		Loop order N.3	
ID	Speedup	T.D.	Speedup	T.D.	Speedup	T.D.
0	1.01	Wo	1.62	Ho	1.02	Ci
1	1.01	Wo+Ci	1.48	Ho	1.01	Wo
2	1.04	Wo	1.66	Ho	1.01	Wo
3	1.00	Wo	2.05	Ho	1.09	Ci
4	1.01	Ci	1.86	Ho	1.02	Ci
5	1.05	Wo	1.82	Ho	1.03	Ci
6	1.07	Wo+Ci	1.91	Ho	1.12	Ci
7	1.04	Wo+Ci	1.80	Ho	1.06	Wo+Ci
8	1.04	Wo+Ci	1.90	Ho	1.58	Ci
9	1.01	Ci	1.77	Ho	1.60	Ci
10	1.02	Ci	1.43	Ho	1.26	Ci
11	1.09	Ci	1.04	Ho	1.60	Ci

(e) Intel, icpc

Layer	Loop order N.1		Loop order N.2		Loop order N.3	
ID	Speedup	T.D.	Speedup	T.D.	Speedup	T.D.
0	1.01	Wo	1.40	Ho	1.01	Wo
1	1.02	Wo	1.25	Ho	1.01	Wo
2	1.16	Wo	1.10	Ho	1.04	Wo
3	1.05	Wo+Ci	1.37	Ho	1.07	Ci
4	1.03	Wo	1.33	Ho	1.04	Wo
5	1.05	Wo	1.38	Ho	1.05	Ci
6	1.19	Wo	1.27	Ho+Wo	1.08	Ci
7	1.05	Wo	1.35	Ho	1.04	Wo+Ci
8	1.14	Wo+Ci	1.34	Ho+Wo	1.15	Wo+Ci
9	1.01	Wo	1.21	Ho+Wo	1.21	Ci
10	1.01	Wo	1.18	Ho	1.10	Ci
11	1.03	Ci	1.11	Ho	1.19	Ci

(f) AMD, icpc

versions. On Intel CPU, the tiling dimension that leads to the best results is always output height (H_o). On AMD CPU, higher speedups are achieved when both output height (H_o) and width (W_o) are used to reduce the workload. The only exception is when g++ is used for compiling, as the best speedup is obtained by applying tiling on output height (H_o) only in most cases. Loop orders N.1 and N.3 benefit from tiling differently depending on the compiler. Using icpc and clang++, the results obtained are quite similar, both in terms of speedup and preferred tiling choice. For them, applying tiling along output width (W_o) is the best choice for initial layers, while intermediate and final layers benefit more from applying tiling on input depth (C_i) and output width (W_o). With g++, the situation is slightly different, as the preferred dimension to apply tiling is always input depth (C_i).

On Intel CPU, tiling helps to increase performance up to 1.11×, 2×, and 1.62× using loop orders N.1, N.2, and N.3, respectively. On AMD CPU, the performance increase is up to 1.21×, 1.41×, and 1.83× using loop orders N.1, N.2, and N.3, respectively.

D. SIMD-VECTORIZATION

We analyze the effects of SIMD-vectorization on each loop order for each compiler. We evaluate the performance of both compiler-driven auto-vectorized code (*auto-simd*) and manually-vectorized code (*simd-8*, *simd-16*, and *simd-32* implemented as described in Section III-C). The baseline version is an implementation in which neither SIMD intrinsics nor compiler auto-vectorization are used (*no-simd*).

When auto-vectorization is disabled (i.e., *no-SIMD* code), clang++ and icpc produce better code than g++ from a performance perspective. As highlighted in Section IV-B (Figure 6), clang++ and icpc have better average execu-

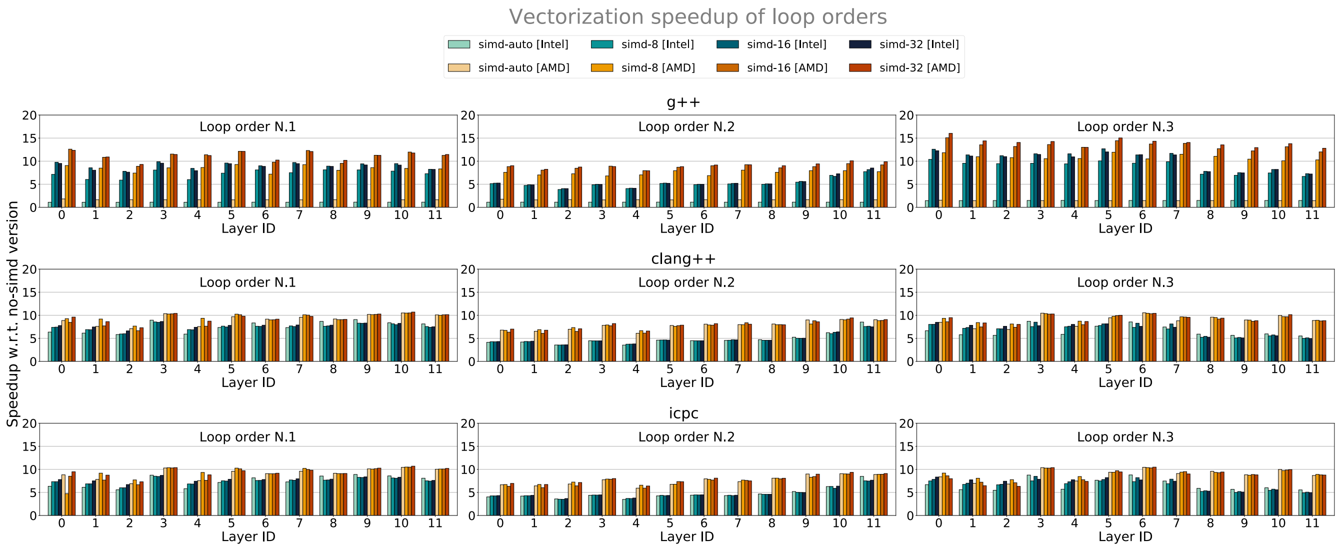


FIGURE 7. Vectorization speedup of selected convolutional layers obtained using loop orders N.1, N.2, and N.3 and compiling with g++, clang++, and icpc. Results are normalized to the version in which auto-vectorization is disabled and no explicit SIMD instructions are inserted.

tion times with respect to g++ on both Intel and AMD CPUs. On Intel CPU, it is 15%, 12%, and 50% better for loop orders N.1, N.2, and N.3, respectively. On AMD CPU, it is 43%, 29%, and 35% better for loop orders N.1, N.2, and N.3, respectively. Figure 7 shows the relative performance achieved by both auto-vectorized and manually-vectorized code, varying loop order and compiler. The results are normalized against their respective *no-simd* version.

1) COMPILER AUTO-VECTORIZATION

g++ is unable to auto-vectorize the code in any of the tested cases. In fact, using clang++ and icpc, *simd-auto* provides at least 4× speedup with respect to *no-simd*, while g++ does not provide any performance improvement on any configuration.

Both clang++ and icpc produce auto-vectorized code with similar performance. We see that auto-vectorization works better on AMD CPU, where the achieved speedup reaches up to 8-10× with all the loop orders, with the exception of the initial layers, where the speedup is slightly lower (5-6×). On Intel CPU, there is more sensitivity to the choice of loop order. The best speedups are achieved with Loop order N.1, ranging from 5× (first layers) to 10× (intermediate and final layers). Loop orders N.2 and N.3 have a speedup that is about 5× on both Intel and AMD CPUs.

2) MANUAL VECTORIZATION

Also with manual vectorization, the results obtained with clang++ and icpc are similar and differ from those obtained with g++.

Compiling with g++, both on Intel and AMD CPUs, the degree of unrolling plays an important role. Using *simd-8* there is a lower speedup, with respect to *simd-auto*, than *simd-16* and *simd-32*. This means that when *simd-8* is used, g++ is not able to fully exploit SIMD units and further manual optimization is needed (i.e., unrolling more instructions in the innermost loop). Using *simd-16* and *simd-32*, performance are, on average, 1.4× better compared to *simd-8*. On both CPUs, *simd-16* and *simd-32* achieve almost the same performance, meaning that unrolling 32 times the innermost loop is not beneficial for g++ compiler.

Using clang++ and icpc, there is no particular variation in performance among *simd-auto*, *simd-8*, *simd-16*, and *simd-32*. They present almost the same speedup, with respect to *no-simd* version, with values ranging from 3.5× to 9× on Intel CPU and from 5× to 10.8× on AMD CPU.

E. PARALLELIZATION STRATEGY

We analyze the effects of parallelization strategies, discussed in Section III-B. SIMD instructions and loop tiling are used in these multi-threaded versions as discussed in Section IV-D and Section IV-C, respectively. In particular, we use intrinsic functions, unrolling loops 32 times, and we apply tiling according to values of Table 4. We study multi-threaded direct convolution using up to 8 threads on the Intel CPU and up to 12 threads on the AMD CPU, so as to exploit the maximum number of physical cores available to each CPU. Moreover, we disable hyperthreading and pin each thread of execution to a physical core in order to have all the core resources available for a single thread.

For the sake of clarity, the following discussion focuses more on the results obtained from versions compiled with g++. For versions compiled with clang++ and icpc, we highlight the most relevant results and the main differences compared to g++ when discussing the best parameter combinations in Section V-A.

Figure 8 shows the speedups of multi-threaded direct convolution implementations obtained by running, individually, all the selected layers, using all loop orders, and all parallelization strategies, on AMD CPU. Compared to the single-thread version, all the parallelization strategies are able to give a 2× speedup when 2 threads are used. This

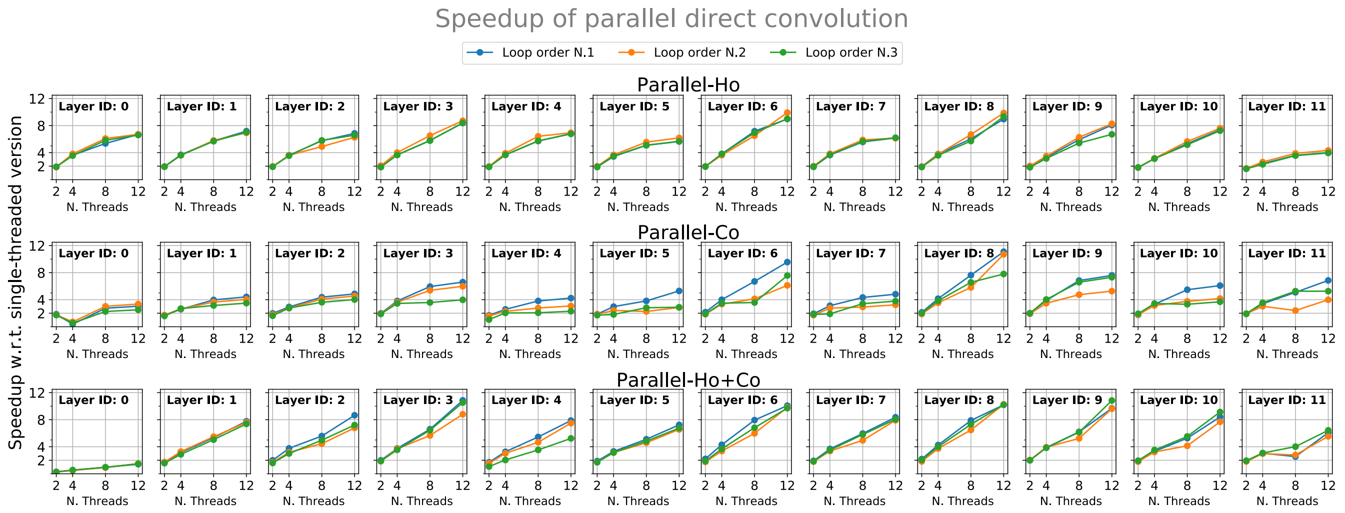


FIGURE 8. Speedup of each multi-threaded direct convolution implementation, obtained running selected convolutional layers and using different loop orders (loop order N.1, loop order N.2, and loop order N.3), different parallelization strategies (parallel-Ho, parallel-Co, and parallel-Ho+Co) and compiling with g++ on AMD CPU. The results are normalized to single-threaded version.

indicates that inter-thread interference is negligible in all the versions analyzed. The only exception occurs with parallel-Ho+Co in the first layer (Layer ID = 0), in which the low number of output channels makes the threads' workload unbalanced.

When the number of threads increases (i.e., 4, 8, and 12 threads), there is an increased sensitivity to the parallelization strategy adopted. In the initial layers, parallel-Co and parallel-Ho+Co do not scale as the number of thread increases. This is due to the low number of output channel (C_o), thus parallel-Ho is preferred.

With 8 threads, some layers (e.g., layer 6) achieve a superlinear speedup due to the better resource utilization that smaller per-core workloads permit. Using 12 threads, there are only few situations in which a 12× speedup is obtained (e.g., layer 8). This phenomenon is expected, since the high number of threads increases the contention of shared resources, i.e., the last level of cache, leading to the non-achievement of ideal linear scaling performance.

Parallel-Ho have a good scalability as long as the spatial dimensions of the output (i.e., height and width) are big enough (initial and intermediate layers). Indeed, when the output height is small (e.g., $H_o = W_o = 9$ in the last layer), the thread workloads are not balanced enough to provide good scalability results. This behaviour is common to every loop order.

Parallel-Co obtains good scalability results especially in the final layers. In the intermediate layers, the performance depends on the type of loop order. Loop orders N.2 and N.3 are not suited for parallel-Co strategy, while order N.1 obtains the better results (up to 75% higher than the other two orders).

Parallel-Ho+Co, like Parallel-Ho, shows a reduced sensitivity to the loop order choice. Intermediate and final layers show a higher scalability.

V. PUTTING IT ALL TOGETHER

After evaluating the effects of loop orders, loop tiling, SIMD-vectorization, and parallelization for each compiler, in this section we show how to combine all the parameters to achieve best performance from direct convolution.

A. BEST COMBINATIONS

In the best combinations, loop tiling and SIMD-vectorization are applied as discussed in Section IV-C and Section IV-D, respectively. We evaluate, after applying loop tiling and SIMD-vectorization, how loop orders and parallelization strategies can be combined together, for each compiler. Figure 9 shows the combinations of loop orders and parallelization strategies that achieve the best performance for each convolutional layer, with different number of threads and compilers. The situation on Intel and AMD CPUs is different.

On Intel CPU, initial layers score the best results using parallel-Ho combined with order N.1 and N.3, for all compilers. In particular, g++ is able to produce better optimization when both order N.1 and N.3 are used, while clang++ and icpc produces better results using only order N.1 and N.3, respectively.

During the execution of intermediate layers, due to the increased tensor depths compared to initial layers, all compilers include parallel-Ho+Co and parallel-Co strategies in their best combinations. In intermediate layers, g++ obtains different best combinations than clang++ and icpc. For g++, these include order N.1 combined with parallel-Ho, and order N.3 combined with parallel-Ho and parallel-Ho+Co for clang++ and icpc.

Running the last layers, the most convenient parallelization strategy is parallel-Co. In these layers, order N.2 is for the first time included in the best combinations of executable code produced by clang++ and icpc, while it is never included in the best combinations of g++.

G=g++ L=clang++ I=icpc 🟦 Intel 🟥 AMD 🟩 Intel & AMD

Layer ID	N. Threads	Parallel-Ho			Parallel-Co			Parallel-Ho+Co		
		N1	N2	N3	N1	N2	N3	N1	N2	N3
0	2	GLI		GLI	G					
	4	GLI		GLI						
	8	GLI		GLI						
	12	GLI								
1	2	GLI		GLI						
	4	GLI		GLI						
	8	GLI		GLI						
	12	G		LI						
2	2	GI		GLI				G		
	4	GI		GLI				G		
	8	GLI		GLI						
	12		I	GLI						
3	2	GLI		LI	GLI			L		GLI
	4	GLI		LI	G		LI			GLI
	8	LI		LI				G		GLI
	12	LI		L						G
4	2	GLI		GLI						
	4	GLI		GLI						
	8	GLI		GLI						
	12	GLI								
5	2	GLI		GLI				L		
	4	GLI		GLI						
	8	GLI		GLI						L
	12			GLI						
6	2				L		L	GLI		LI
	4							GLI		LI
	8							GLI		LI
	12							GLI		
7	2	GLI		L			LI	G		LI
	4	GLI		L				G		LI
	8	LI		L				GLI		LI
	12	LI						G		
8	2				GLI		LI	GLI		LI
	4				G			GLI		LI
	8							GLI		L
	12				G			LI		
9	2	LI			GLI			GLI		
	4				GLI			GLI		
	8	LI			GI	LI		GI		
	12				G			LI		
10	2	GLI			GLI			GLI		
	4	LI			GLI			GLI		
	8	GLI			G	LI		GLI		
	12	LI	L		G					
11	2				GI	LI		G		
	4				GLI			LI		
	8		LI		GLI	LI		I		
	12				G			LI		

FIGURE 9. **Summary of the best parameter combinations. For each row, letters *G*, *L*, and *I* indicate the best parameter combination achievable compiling with g++, clang++, and icpc, respectively, on Intel CPUs (blue) and AMD CPUs (red). If the same letter is repeated more than once in the same row, it means that two or more combinations achieve the same performance. As a consequence, blank cells identify configurations that are surpassed by others performance-wise. Considering the underlying machine, the compiler of choice, and the number of threads (as much as physical cores is the best), a practical use of this table by a user would be the following: for each layer, locate in the corresponding row the letter of the compiler of choice in the coloring that corresponds to the underlying CPU (green, otherwise red for AMD and blue for Intel). That is the best-performing configuration, so parallelization-strategy and loop order must be selected accordingly. 12-threads configurations are not taken into account for Intel CPU because it only has 8 physical cores.**

On AMD CPU, initial layers achieve the best results using parallel-Ho combined with order N.1 and N.3, for all compilers. In particular, g++ is able to produce better optimization when both order N.1 and N.3 are used (as on Intel CPU), while clang++ and icpc produce better results with orders N.3 and N.1, respectively (this is the opposite situation than Intel CPU).

Executing intermediate layers, parallel-Ho and parallel-Ho+Co are the predominant best parallelization strategies, while parallel-Co appears only in some best combinations (i.e., layer IDs 3 and 8).

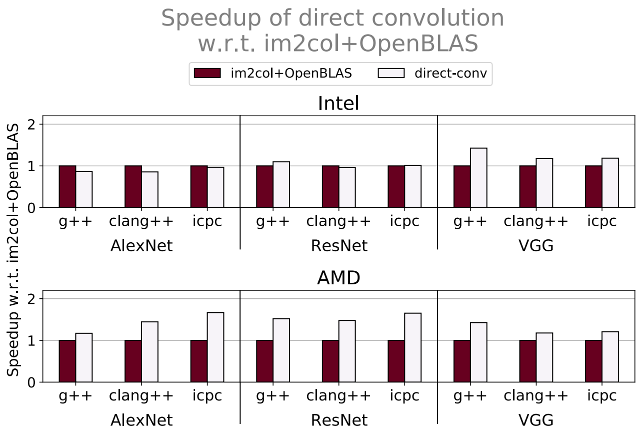
As in the Intel CPU case, order N.2 is included in the best combinations only in the last layers. In particular, clang++ and icpc include order N.2 in their best combinations when parallel-Ho+Co and parallel-Ho are used.

B. OPTIMIZATION HEURISTICS

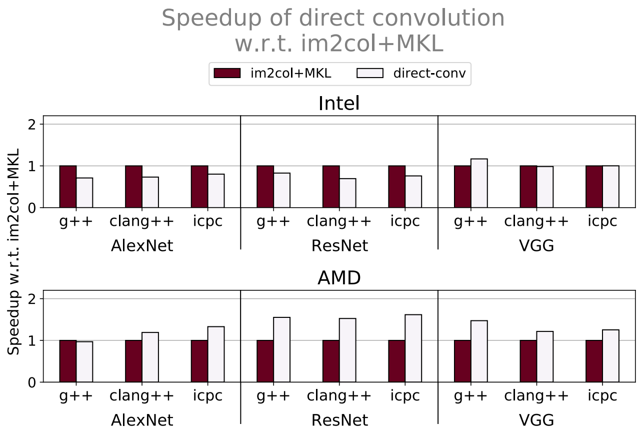
From the per-parameter analysis conducted in the previous section and the best combinations highlighted in Subsection V-A, we can derive some heuristics to help direct convolution implementors to easily select a high-performance implementation for the system at hand. We suggest adopting a parametric implementation as we do in this paper and select a configuration by following the steps suggested below. Applying these heuristics might not necessarily lead to the optimal configuration for a given system, but the results of the extensive analysis performed should make them able to suggest a *good enough* configuration, or a starting point for conducting more detailed parameter searches without needing to explore the whole parameter space.

In the following, we list some heuristics to select a high-performance configuration for a parametric implementation of a direct convolution on a multi-core CPU:

- Parameter tuning should be done on a per-layer basis rather than a per-network one.
- A parallel implementation should be preferred, with as many threads as physical cores.
- For the first layers of a CNN, generally characterized by input with high spatial dimensions (i.e., height and width), select parallelization strategy parallel-Ho if clang++ or icpc is the compiler of choice, and parallel-Ho+Co if it is g++. Select loop order N.1 or N.3, and prefer the latter for processors with small L2 caches (e.g., Intel CPU in this paper).
- For intermediate and final layers of a CNN, generally characterized by input with low spatial dimensions but with high number of channels (i.e, depth), select parallelization strategy parallel-Co or parallel-Ho+Co. Select loop order N.3 on processors with small L2 caches (i.e., less than 256 KB), and loop order N.1 otherwise.
- Use loop tiling, especially on processors with smaller L2 caches. Apply it along output width (Wo) and input depth (Ci) dimensions if loop orders N.1 or N.3 have been selected, apply it along output height (Ho) otherwise.
- Use SIMD-vectorization. Prefer manually-inserted SIMD instructions (i.e., *simd-8*/*simd-16*/*simd-32*) to



(a) Direct convolution speedup w.r.t. im2col+OpenBLAS



(b) Direct convolution speedup w.r.t. im2col+IntelMKL

FIGURE 10. **Speedup of direct convolution, implemented according to heuristics discussed in Section V-B, respect to (a) im2col+OpenBLAS and (b) im2col+MKL. Results are obtained compiling with g++, clang++ and icpc, running AlexNet, ResNet and VGG on both Intel and AMD CPUs. Direct convolution results are obtained by using the best combination for each layer. Both im2col+gemm and direct convolution are executed using the maximum number of available threads (i.e., 8 threads on Intel CPU and 12 threads on AMD CPU).**

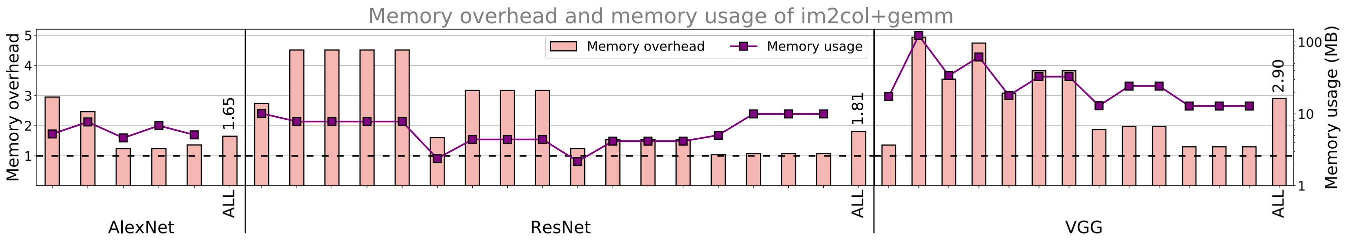


FIGURE 11. **Memory overhead and memory usage of *im2col+gemm*. Memory overhead is expressed as the ratio between the total memory used by *im2col+gemm* and the memory used by direct convolution (i.e., the memory needed to store tensors). Memory usage is expressed as the total memory used (MB) to store all tensors of each convolutional layer. Results are shown for convolutional layers of AlexNet, ResNet, and VGG.**

compiler auto-vectorization (i.e., *simd-auto*). If the compiler of choice is g++, *simd-32* is the best choice.

C. im2col+gemm COMPARISON

Figures 10a and 10b show the performance achieved by a direct convolution, implemented according to heuristics discussed in Section V-B, compared against an *im2col+OpenBLAS* and *im2col+MKL*, respectively. *im2col+gemm* results are obtained using the maximum number of threads on each CPUs, (i.e., 8 and 12 threads on Intel and AMD CPUs, respectively). The speedup of direct convolution with respect to *im2col+gemm* depends on underlying CPU and BLAS library provider.

On AMD CPU, direct convolution implementation allows having a speedup that ranges from 1.17× to 1.67× over *im2col+OpenBLAS* and a speedup in the range 0.98×–1.62× over *im2col+MKL*.

On Intel CPU, direct convolution achieves better, or comparable, results than im2col+OpenBLAS, with speedups that range from 0.95× to 1.4×. Compared to *im2col+MKL*, conversely, direct convolution results are better only compiling with g++ and using VGG network (1.15× speedup). This is an expected result, since the matrix-multiplication routine is highly optimized on Intel CPUs by Intel-MKL library.

In general, the results of direct convolution obtained on AMD CPU are better, compared to *im2col+gemm* versions, than Intel CPU. This indicates that, possibly, the bigger L2 and L3 caches of AMD CPU allow better exploiting data locality exposed by our direct convolution implementations through the selection of the best memory access pattern and tiling strategy. Since the trend is to have bigger caches, we can expect that the advantage of a direct convolution over an *im2col+gemm* implementation could be destined to increase.

Another aspect worth of an analysis is the memory occupancy of direct convolution and im2col. Figure 11 shows the memory overhead of *im2col+gemm* methods. The total memory overhead of im2col+gemm is 1.65×, 1.81×, and 2.90× for AlexNet, ResNet, and VGG, respectively. For this reason, a direct convolution implementation is more palatable from a memory occupancy perspective, and should be considered when a limited memory footprint is required.

Moreover, the only case in which direct convolution achieves better performance than *im2col+MKL* on Intel machine is when a VGG network is executed. Figure 11 shows that VGG is the network with higher memory overhead. This datum suggests that, when there is a high memory overhead, a direct convolution implementation could be beneficial. However, in some cases im2col+gemm shows better performance than direct convolution (e.g., AlexNet executed with *im2col+MKL* on an Intel machine). This implies that the latencies induced by tensor transformation and a higher memory footprint are hidden by the performance gain of optimized matrix-matrix multiplication routines.

VI. RELATED WORK

There are several prior works that aim to improve the conventional *im2col+gemm* implementation. They can be grouped in two macro categories.

Optimized im2col+gemm. This category includes proposals [12], [24], [25], [26] that optimize *im2col+gemm* implementation using specific optimized *gemm* kernels and/or by reducing the expensive memory overhead. The authors of [12] propose a convolution implementation specifically for ARM architectures (especially present in mobile devices) based on the key-idea that, unlike x86 architectures, high performance convolution implementations are bottlenecked by the cache-to-register memory transfers. Most of the works in this category [24], [25], [26] propose to reduce im2col memory overhead accessing data using specific patterns that increase data locality and data reuse. The use of blocked computation is generally adopted [24], [25], [26] to address *gemm* operations.

Optimized direct convolution. Proposals [11], [13], [27], [28], [29], [30] belong to this category. They are characterized by direct convolution implementations obtained through manual optimizations and/or parameters tuning. High performance are achieved exploiting convolution memory access patterns and underlying architecture. In [11], the authors propose a loop order and related memory layouts that can be parameterized with architectural characteristics. A work [27] proposes two direct convolution implementations targeting ARM architecture: 1) one with a zero-memory overhead and 2) one relying on a small buffer that is used to speed up the computation. An optimized convolution operator, specifically for x86 architectures, is presented in [13]. The latter targets convolution used in most popular image processing operations (e.g., Gaussian blur). The proposal leverages on most common high performance techniques such as SIMD instructions and multi-threading. Authors of [13] refer to future work on implementing convolution operation for CNNs. Some prior works [28], [29], [30] explore the entire convolution design space. Using this type of methods, the goal is to minimize certain values (e.g., data movement [28]) by solving analytical models.

Our work falls into the second category (i.e., optimized direct convolution), since we extend direct convolution basic implementation exploiting high performance techniques such as loop orders, loop unrolling, loop tiling, SIMD instructions, and multi-threading. Our proposals differs from the others as we explore several implementation parameter and study how they combine with each other. Another aspect worth mentioning is that the authors of [11] propose a specific blocked layout to store tensors in memory that needs to be adapted for each convolutional layer. Authors of [27], as in our proposal, show the benefits of SIMD instructions and how they can be used in direct convolution micro-kernel implementations. The latter presents and discuss results obtained using ARM architecture in single-threaded implementations.

VII. CONCLUSION AND FUTURE WORK

The analysis done in this work highlights that *direct convolution* can be a promising approach to address convolutions in CNNs, and have the potential to replace the matrix-matrix multiplication-based approach, which is currently the most common. However, direct convolution can be implemented based on few parameters that must be tuned to achieve the best performance. We identify four parameters: loop organization (i.e., loop orders and loop tiling), parallelization strategy, SIMD-vectorization, and compiler choice.

We explored different combinations of these parameters on two real multi-core systems, analyzing the achieved performance of each parameter set on different architectures and with convolutional layers with different dimensions. We identified some heuristics that can help direct-convolution implementers achieving high performance, that can be summarized as: tune parameters on a per-layer basis, use a parallel implementation with as many threads as physical cores, use loop tiling, use SIMD-vectorization with manually-inserted SIMD instructions. For initial layers, parallelize on output height when compiling with clang++/icpc and also on output channels with g++, with loop orders N.1 or N.3. For intermediate and final layers, parallelize on both output height and channels, select loop order N.3 if the L2 cache is small, N.1 otherwise.

The results of this work can help convolution designers to move away from the usual approach based on matrix-matrix multiplication and embrace direct convolution, which can deliver better performance with a reduced memory footprint. For this purpose, we provide some heuristics to adopt to achieve a direct convolution implementation that scores high performance on a given architecture.

As future work, we plan to enrich the current study with an additional parameter: tensor memory layout. By including this parameter and enlarging the set of architectures considered for the study, we plan to improve our heuristics to include a bigger set of possibilities. We also plan to extend our results in the context of ad-hoc hardware accelerators. In particular, we plan to investigate how our findings can help accelerator designers to implement the data-flow in hardware.

REFERENCES

[1] S. S. Farfade, M. J. Saberian, and L.-J. Li, "Multi-view face detection using deep convolutional neural networks," in *Proc. 5th ACM Int. Conf. Multimedia Retr.*, Jun. 2015, pp. 643–650.

[2] S. Li, Z. Liu, and A. B. Chan, "Heterogeneous multi-task learning for human pose estimation with deep convolutional neural network," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. Workshops*, Jun. 2014, pp. 488–495.

[3] K. Lan, D.-T. Wang, S. Fong, L.-S. Liu, K. K. L. Wong, and N. Dey, "A survey of data mining and deep learning in bioinformatics," *J. Med. Syst.*, vol. 42, no. 8, pp. 1–20, Aug. 2018.

[4] W. Wang and J. Gang, "Application of convolutional neural network in natural language processing," in *Proc. Int. Conf. Inf. Syst. Comput. Aided Educ. (ICISCAE)*, Jul. 2018, pp. 64–70.

[5] N. Kalchbrenner, E. Grefenstette, and P. Blunsom, "A convolutional neural network for modelling sentences," 2014, *arXiv:1404.2188*.

[6] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, 2012, pp. 1–11.

[7] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," 2014, *arXiv:1409.1556*.

[8] A. Khan, A. Sohail, U. Zahoora, and A. S. Qureshi, "A survey of the recent architectures of deep convolutional neural networks," *Artif. Intell. Rev.*, vol. 53, no. 8, pp. 5455–5516, Dec. 2020.

[9] (2015). *OpenBLAS*. [Online]. Available: http://www.openblas.net/

[10] Intel. (2015). *Math Kernel Library*. [Online]. Available: https://software.intel.com/en-us/intel-mkl

[11] J. Zhang, F. Franchetti, and T. M. Low, "High performance zero-memory overhead direct convolutions," in *Proc. Int. Conf. Mach. Learn.*, 2018, pp. 5776–5785.

[12] P. Zhang, E. Lo, and B. Lu, "High performance depthwise and pointwise convolutions on mobile devices," in *Proc. AAAI Conf. Artif. Intell.*, Apr. 2020, vol. 34, no. 4, pp. 6795–6802.

[13] V. Kelefouras and G. Keramidas, "Design and implementation of 2D convolution on x86/x64 processors," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 12, pp. 3800–3815, Dec. 2022.

[14] J. Guo, R. Teodorescu, and G. Agrawal, "Fused DSConv: Optimizing sparse CNN inference for execution on edge devices," in *Proc. IEEE/ACM 21st Int. Symp. Cluster, Cloud Internet Comput. (CCGrid)*, May 2021, pp. 545–554.

[15] B. Peccerillo, M. Mannino, A. Mondelli, and S. Bartolini, "A survey on hardware accelerators: Taxonomy, trends, challenges, and perspectives," *J. Syst. Archit.*, vol. 129, Aug. 2022, Art. no. 102561.

[16] P. Chen, M. Wahib, S. Takizawa, R. Takano, and S. Matsuoka, "A versatile software systolic execution model for GPU memory-bound kernels," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, New York, NY, USA, Nov. 2019, pp. 1–81.

[17] P. A. Martínez, B. Peccerillo, S. Bartolini, J. M. García, and G. Bernabé, "Performance portability in a real world application: PHAST applied to Caffe," *Int. J. High Perform. Comput. Appl.*, vol. 36, no. 3, pp. 419–439, May 2022.

[18] Y. Chen, T. Yang, J. Emer, and V. Sze, "Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices," *IEEE J. Emerg. Sel. Topics Circuits Syst.*, vol. 9, no. 2, pp. 292–308, Jun. 2019.

[19] Y. Chen, T. Chen, Z. Xu, N. Sun, and O. Temam, "DianNao family: Energy-efficient hardware accelerators for machine learning," *Commun. ACM*, vol. 59, no. 11, pp. 105–112, Oct. 2016.

[20] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2016, pp. 770–778.

[21] H. Jeong, S. Kim, W. Lee, and S.-H. Myung, "Performance of SSE and AVX instruction sets," 2012, *arXiv:1211.0820*.

[22] J. Kukunas, *Power and Performance: Software Analysis and Optimization*. San Mateo, CA, USA: Morgan Kaufmann, 2015.

[23] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," 2014, *arXiv:1408.5093*.

[24] T. Zhao, Q. Hu, X. He, W. Xu, J. Wang, C. Leng, and J. Cheng, "ECBC: Efficient convolution via blocked columnizing," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 34, no. 1, pp. 433–445, Jan. 2021.

[25] M. Cho and D. Brand, "MEC: Memory-efficient convolution for deep neural network," in *Proc. Int. Conf. Mach. Learn.*, 2017, pp. 815–824.

[26] M. Dukhan, "The indirect convolution algorithm," 2019, *arXiv:1907.02129*.

[27] S. Barrachina, A. Castelló, M. F. Dolz, T. M. Low, H. Martínez, E. S. Quintana-Ortí, U. Sridhar, and A. E. Tomás, "Reformulating the direct convolution for high-performance deep learning inference on ARM processors," *J. Syst. Archit.*, vol. 135, Feb. 2023, Art. no. 102806.

[28] R. Li, Y. Xu, A. Sukumaran-Rajam, A. Rountev, and P. Sadayappan, "Analytical characterization and design space exploration for optimization of CNNs," in *Proc. 26th ACM Int. Conf. Architectural Support Program. Lang. Operating Syst.*, Apr. 2021.

[29] X. Yang, J. Pu, B. B. Rister, N. Bhagdikar, S. Richardson, S. Kvatinsky, J. Ragan-Kelley, A. Pedram, and M. Horowitz, "A systematic approach to blocking convolutional neural networks," 2016, *arXiv:1606.04209*.

[30] X. Zhang, J. Xiao, and G. Tan, "I/O lower bounds for auto-tuning of convolutions in CNNs," 2020, *arXiv:2012.15667*.

MIRCO MANNINO received the B.Sc. and M.Sc. degrees in computer engineering from the University of Siena, where he is currently pursuing the Ph.D. degree with the Department of Information Engineering and Mathematical Sciences. His research interests include the optimization of deep learning algorithms and parallel algorithms, hardware accelerators, and virtual memory.

BIAGIO PECCERILLO is currently a Postdoctoral Researcher with the Department of Information Engineering and Mathematical Sciences, University of Siena. His research interests include heterogeneous architectures, productivity-oriented high-level abstraction mechanisms, hardware accelerators, and parallel algorithms. He has participated in various research and development projects involving high-productivity solutions to program heterogeneous architectures, hardware accelerators, haptic algorithms in virtual and augmented reality environments, and pharmaceutical supply chain simulation.

ANDREA MONDELLI received the Ph.D. degree in computer architecture. He is currently the CPU Chief Architect with Huawei and a Principal Researcher of cybersecurity and architecture design. He is also a Technology Manager and responsible for Huawei research projects collaborations with European universities. He was a researcher and an architect in various countries, such as Italy, USA, France, China, and U.K. He has published multiple manuscripts and conference papers and a book on memory coherence protocols. His research interests include high performance and low power CPUs. He was a part of RISC-V International as the Chair of Virtual Memory.

SANDRO BARTOLINI is currently an Associate Professor with the Department of Information Engineering and Mathematical Sciences, University of Siena. His research interests include high-performance chip multiprocessors (CMPs), the new approaches to productive programming of heterogeneous architectures (CPUs and GPUs), integrated photonics for CMPs, feedback-driven compiler optimizations for cache hierarchy performance and low power, and hardware accelerators. He has led and participated in various research and development projects. He is an Associate Editor of the *EURASIP Journal of Embedded Computing* and an active member of the HiPEAC NoE. He has been the Co-Guest Editor of *Transactions on High Performance Architectures and Compilation* (Springer) journal and ACM SigArch Computer Architecture Newsletter.

Open Access funding provided by 'Università degli Studi di Siena' within the CRUI CARE Agreement