GEMM-Like Convolution for Deep Learning Inference on the Xilinx Versal

Jie Lei[1], Héctor Martínez[2], José Flich[1], and Enrique S. Quintana-Ortí[1]

[1] Universitat Politècnica de València, Spain.
{jlei,jflich,quintana}@disca.upv.es
[2] Universidad de Córdoba, Spain. el2mapeh@uco.es

Abstract. We revisit a blocked formulation of the direct convolution algorithm that mimics modern realizations of the general matrix multiplication (GEMM), demonstrating that the same approach can be adapted to deliver high performance for deep learning inference tasks on the AI Engine (AIE) tile embedded in Xilinx Versal platforms. Our experimental results on a Xilinx Versal VCK190 shows an arithmetic throughput close to 70% of the theoretical peak of the AIE tile for 8-bit integer operands and the convolutional layers arising in ResNet-50 v.15+ImageNet.

Keywords: Deep learning · Convolution · Direct algorithm · High Performance · SIMD units · Cache Memory.

1 Introduction

The convolution (CONV) is a key operator for the type of deep neural networks (DNNs) that dominate machine learning for signal processing, including computer vision tasks [4,12,13]. For this reason, during the last years there has been an intensive effort to carefully exploit the architecture of modern processors in order to produce efficient realizations of the operator via either the classical algorithm for the direct convolution [3, 16], the lowering (or "im2col") approach based on the matrix multiplication (GEMM) [2, 6], the Fast Fourier transform (FFT)-based convolution, or Winograd's minimal filtering algorithms [7, 10, 17].

In this paper, we revisit the GEMM-like algorithms for the direct convolution introduced in [3, 16], exploring how to efficiently map them to the Artificial Intelligence Engine (AIE) in the Xilinx Versal accelerator. In doing so, we make the following specific contributions:

– We offer a practical demonstration that the same ideas underlying the high performance realization of the direct convolution on conventional processors, equipped with SIMD (single instruction, multiple data) units and multi-layered memories, carry over to the AIE-enabled Xilix Versal ACAP.
– We customize our general design of the direct convolution for the Xilinx Versal VCK190, conducting a complete experimental analysis for a representative convolutional neural network (CNN): ResNet-50 v1.5+ImageNet.

The rest of the paper is structured as follows. In Section 2 we expose the connection between the high performance realizations of the CONV operator and GEMM. In Section 3 we present our strategy to map CONV on the Xilinx Versal VCK190, and in Section 4 we evaluate the approach. Finally, in Section 5 we close the paper with a few remarks and a discussion of future work.

2 Blocking in the Direct Convolution

In this section we follow [3] in order to expose the connection between the convolution operator and the high performance implementations of GEMM in modern instances of the BLAS (Basic Linear Algebra Subprograms) [8].

2.1 High performance GEMM in conventional processors

```
for (jc=0; jc<n; jc+=nc) // Loop L1
 for (pc=0; pc<k; pc+=kc){     // L2
  // Pack B
  Bc:=B(pc:pc+kc-1, jc:jc+nc-1);
  for (ic=0; ic<m; ic+=mc){    // L3
   // Pack A
   Ac:=A(ic:ic+mc-1, pc:pc+kc-1);
   for (jr=0; jr<nc; jr+=nr)   // L4
     for(ir=0; ir<mc; ir+=mr)  // L5
      // Micro-kernel
      C(ic+ir:ic+ir+mr-1,
        jc+jr:jc+jr+nr-1)
        += Ac(ir:ir+mr-1, 0:kc-1)
        *  Bc(0:kc-1, jr:jr+nr-1);
 }}
```

```
for (pr=0; pr<kc; pr++) // L6
  C(ic+ir:ic+ir+mr-1,
    jc+jr:jc+jr+nr-1)
    += Ac(ir:ir+mr-1,pr)
    *  Bc(pr,jr:jr+nr-1);
```

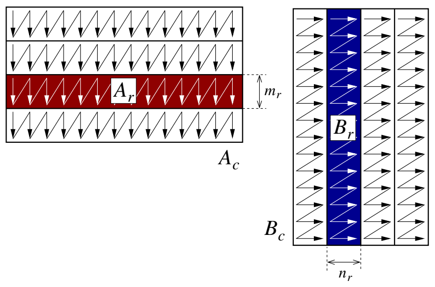

Fig. 1: High performance algorithm for GEMM. Top-Left: Blocked algorithm; Top-Right: Micro-kernel; Bottom-Right: Packing.

Consider the GEMM $C = A \cdot B$, where $A \rightarrow m \times k$, $B \rightarrow k \times n$, and $C \rightarrow m \times n$. The modern implementations of this kernel (e.g., in AMD AOCL, OpenBLAS, BLIS, etc.) follow GotoBLAS2 [9] to formulate it as a blocked algorithm comprising five nested loops around two packing routines and an architecture-dependent *micro-kernel*; see Figure 1, top-left.

For the GEMM algorithm in the figure, an appropriate choice of the strides for the three outermost loops, given by m_c, n_c, k_c, (and known as the *cache configuration parameters*,) combined with a certain packing of the matrix inputs into two buffers, $A_c \rightarrow m_c \times k_c$ and $B_c \rightarrow k_c \times n_c$, (Figure 1, bottom-right), orchestrates a careful pattern of data movements across the memory hierarchy that reduces the number of number of cache misses [11,14]. Hereafter we assume that m, n, k are integer multiples of m_c, n_c, k_c, respectively.

The micro-kernel comprises a sixth loop (`L6` in Figure 1, top-right) that performs a sequence of k_c rank-1 updates on an $m_r \times n_r$ *micro-tile* of C, denoted as C_r, each involving a column of an $m_r \times k_c$ micro-panel of the packed buffer A_c and a row of a $k_c \times n_r$ micro-panel of the packed buffer B_c (blocks A_r and B_r in Figure 1, bottom-right.) For processors with SIMD arithmetic units, one of the micro-kernel "dimensions" is chosen to perform the accumulation on C_r using SIMD arithmetic instructions, and the special packing of the elements of A, B into the buffers A_c, B_c enables loading their data using SIMD instructions.

2.2 The Convolution Operator

Consider now a CONV operator, $O = \text{CONV}(F, I)$, that applies a 4D filter tensor $F \rightarrow c_i \times h_f \times w_f \times c_o$, on a 4D input tensor $I \rightarrow b \times h_i \times w_i \times c_i$, to produce a 4D output tensor $O \rightarrow b \times h_o \times w_o \times c_o$. Here, b denotes the batch size (i.e., the number of samples); $c_i | c_o$ stand for the number of input|output channels; $h_i \times w_i | h_o \times w_o$ are the input|output height $\times$ width; and $h_f \times w_f$ are the filter height $\times$ width. Furthermore, assuming a padding p along the dimensions h_i and w_i, then $h_o = \lfloor (h_i - h_f + 2p)/s + 1 \rfloor$ and $w_o = \lfloor (w_i - w_f + 2p)/s + 1 \rfloor$.

A simple algorithm that computes the convolution is displayed in Figure 2. For deep learning (DL) applications, the input and output tensors are generally arranged in memory following either the NHWC or the NCHW layouts, where N specifies the batch dimension; H|W refer to the image height|width; and C correspond to the input|output channels. Also, the filter tensor is stored in either the CRSK (for NHWC) or KCRS (for NCHW) format, where C|K specify the input|output channels, and R|S denote the filter height|width.

```
void ConvDirect(I[b][hi][wi][ci], F[ci][hf][wf][co], O[b][ho][wo][co]){
for (h=0; h<b; h++)
 for (i=0; i<ci; i++)
  for (j=0; j<co; j++)
   for (k=0; k<wo; k++)
    for (l=0; l<ho; l++)
     for {m=0; m<wf; m++)
      for {n=0; n<hf; n++)
       O[h][l][k][j] += I[h][l+n][k+m][i] * F[i][n][m][j];
}
```

Fig. 2: Simple algorithm for the direct convolution.

2.3 The direct convolution is GEMM in disguise

In essence, the realization of the direct convolution in Figure 2, consisting of 7 nested loops around a MAC (multiply-and-add) operation, features the same scheme that is present in the naive implementation of GEMM consisting of three nested loops ijk around a MAC. We next revisit [3] to review that, in the case of the direct convolution, it is possible to apply the same techniques that transform the naive GEMM algorithm into a high performance blocked realization.

As a starting point, consider the NHWC layout for the input/output tensors and the filter tensor arranged in an "alternative" RSCK layout. In [3], the work in [16] is extended to reorganize the simple algorithm for the direct convolution following the principles for the optimization of GEMM in current architectures:

– Select the inner loops to saturate computations via a proper micro-kernel.
– Reorder the outer loops and block the operands to optimize data reuse.
– Apply a cache-aware layout of the operands.
– Decouple the microkernel dimension from the cache blocking parameters.

The application of these principles to the simple algorithm for the direct convolution result in the reformulated variant in Figure 3, which exhibits the following properties [3,16]:

– It preserves the NHWC layout for the input/output tensors.
– It avoids stalls in the micro-kernel due to consecutive writes to the same entry of the output tensor, prevents register spilling and, when possible, accommodates SIMD arithmetic and loads/stores by ensuring access to the convolution operators with unit stride;
– It reduces the memory access overhead by splitting the tensors into smaller blocks that fit into the memory hierarchy; and
– It exposes sufficient thread-level parallelism.

Note the different ordering of the loops for the GEMM and the blocked convolution: L1⇒L2⇒L3⇒L4⇒L5 for the former vs L3⇒L2⇒L1⇒L5⇒L4 for the latter. In addition, the packings of $A|B$ into $A_c|B_c$ occur in loops L3|L2 in GEMM while, for the blocked convolution, if necessary, $I|F$ should be packed into two buffers $I_c|F_c$ in loops L2|L3. These changes aim at maintaining a buffer I_c, of dimension $w_{o,b} \times c_{i,b}$ in the L3 cache, and a buffer F_c, of dimension $c_{i,b} \times c_{o,b}$ in the L2 cache. This is referred to as the A3B2C0 variant of GEMM in [5].

2.4 Packing in the direct convolution

The GEMM-oriented algorithm for the direct convolution assumes that the filter tensor is arranged in a manner that resembles the packing that is required for the micro-panel of B_c in the GEMM algorithm. For DL inference the filter tensor can be pre-packed and stored in the corresponding memory level. The cost of this transformation is amortized for all inference samples and becomes negligible.

With respect to the packing of the input tensor, the GEMM-oriented algorithm for the direct convolution retrieves the entries of I, from the micro-kernel, with a non-unit stride. Fortunately, it is possible to ensure access to I with unit stride from the micro-kernel by packing the entries of I into the $w_{o,b} \times c_{i,b}$ buffer I_c inside loop L2 in Figure 3; see also [3]. This is equivalent to the packing scheme for the $m_c \times k_c$ buffer A_c in the GEMM algorithm.

```
void ConvDirect_GEMM(I[b][hi][wi][cib], F[hf][wf][ci][co],
                     O[b][ho][wo][cob]){
for (h=0; h<b; h++)
 for (l=0; l<ho; l++)
  for (n=0; n<hf; n++)
   for (m=0; m<wf; m++)
    for (kp=0; kp<wo; kp+=wob)     // Loop L3, wo = m
     for (ip=0; ip<ci; ip+=cib)         // L2, ci = k
      for (jp=0; jp<co; jp+=cob)        // L1, co = n
       for (ir=0; ir<wob; ir+=mr)       // L5, wob = mc
        for (jr=0; jr<cob; jr+=nr){     // L4, cob = nc
         ic = kp+ir; jc = jp+jr; kc = ip;
         // Micro-kernel
         for (pr=0; pr<cib; pr++)       // L6, cib = kc
          O[h][l][ic:ic+mr][jc:jc+nr] += I[h][l+n][ic+m:ic+m+mr][kc+pr]
                                       *  F[n][m][kc+pr][jc:jc+nr];
}       }
```

Fig. 3: GEMM-like reformulation of the direct convolution. The comments identify the relationship with the loops inside the GEMM algorithm in Figure 1.

To summarize the previous discussion, the direct convolution can be blocked applying some of the same techniques that are present in modern high performance realizations of GEMM.

3 Mapping CONV to the Xilinx Versal VCK190

In this section, after a brief description of the Xilinx platform, we present our strategy to map the CONV operator to this system.

3.1 Xilinx Versal VCK190 ACAP

The Versal VC1902 processor [1] comprises, as main components, (1) a dual-core ARM Cortex-A72 processor plus a dual-core ARM Cortex-R5F processor; (2) a customizable FPGA with 899,840 LUTs (look-up tables); and (3) 400 standalone AIE tiles organized in a 2D array. Furthermore, each AIE tile contains a SIMD arithmetic unit, providing up to 128 MAC operations per clock cycle for (unsigned) integer 8-bit (UINT8) arithmetic.

In addition, the Versal VCK190 features a versatile memory with 32 KB of “local” memory per AIE tile; distributed Block and Ultra RAMs with capacity for 4.3 MB and 16.3 MB, respectively; and a global 2-GB DDR4 memory.

3.2 Distributing the data across the memory hierarchy

Our data mapping strategy leverages the cache-friendly blocked algorithm for CONV in Figure 3, combined with the following distribution of the problem data across the levels of the memory hierarchy (see also Figure 4):

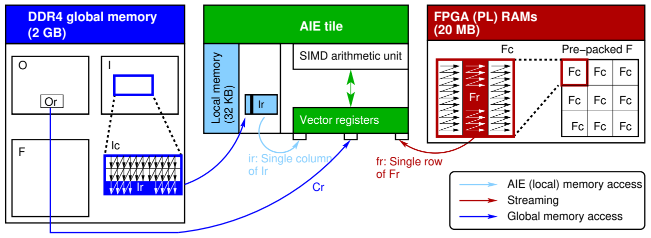

Fig. 4: Data mapping and transfers across the Versal VCK190 memory hierarchy. The transfers from O_r in the global memory, F_r in the FPGA memory, and I_r in the local memory move data directly to the vector registers. The data copy of the micro-panel I_r, from the global memory into the local memory, is carried out by one of the scalar engines (ARM processors).

– The operand F contains the read-only filters for the convolution. The full matrix can be pre-packed off-line into a collection of buffers F_c and kept in the FPGA RAMs, contributing a null cost to the inference process as the same filters are used for many samples.
– Matrix I corresponds to the activation inputs of the CONV operator. This operand is partitioned and packed into a buffer I_c during the execution of CONV in the DDR4 global memory. Also, each individual micro-panel I_r is copied into the AIE local memory. The cost of transferring I_r is amortized by re-using its entries across several iterations of loop L5 in Figure 1, top-left.
– Matrix O contains the activation outputs of the CONV operator. At each execution of the micro-kernel, a small micro-tile O_r, of dimension $m_r \times n_r$, is first loaded directly from the DDR4 global memory into the AIE tile vector registers, and written back to memory at the end; see Figure 1, top-right.

At this point, it is worth remarking some differences between our solution and the approach taken when implementing GEMM in a conventional processor:

– From the point of view of hardware, a conventional processor integrates a number of cache levels (usually, between two and three), and relies on a memory controller to orchestrate the data movements across them. In contrast, in the Versal VCK190 the transfers need to be explicitly encoded into the algorithm. This puts an extra burden into the programmer's shoulders, but offers a strict control over the data transfers.
– The Versal system provides streaming access methods for communication with the "outside world." This provides high throughput and lower latency for the FPGA to AIE communication, but is also fundamentally different from the type of communication that occurs between the arithmetic units and the cache/memory levels in a conventional processor.
– From the algorithmic point of view, given that the target is to implement GEMM for DL inference, we can pre-pack the weight/filter matrix F into a collection of buffers residing into the FPGA RAM. Therefore, the packing that occurs within loop L3 of the baseline algorithm for GEMM is unnecessary.

3.3 Design of the micro-kernel

Following the general trend toward leveraging low precision arithmetic for DL inference, we selected UINT8 as the baseline datatype for the implementation of the CONV operator in the Versal VC1902. Furthermore, given the capacity and number of accumulator registers in the AIE tile, we set the dimensions of the micro-kernel to $m_r \times n_r = 8 \times 8$; that is, the micro-tile O_r updated inside loop `L6` of the micro-kernel comprises 8 entries of 8 columns of O.

Figure 5 displays our micro-kernel for the VC1902. After a few declarations and initializations, the code comprises a loop (Line 18), corresponding to `L6`, that iterates over the $c_{i,b}$ dimension of the micro-panels I_r, F_r, with an unrolling factor of 16. At each iteration, the loop body multiplies the entries in 16 columns of I_r (128 elements, retrieved in `ir0` and `ir1`) with those in 16 rows of F_r (128 elements, in `fr`), accumulating the intermediate results on four different registers via eight invocations to the AIE intrinsic `mac16()` (in Lines 29–42).

Each call to `mac16()` computes 128 UINT8 MAC operations in one cycle, operating with two vectors: one with 64 elements and the other with 32 elements. As `v8uint8` is not supported by the AIE intrinsics, we concatenate two column vectors of the micro-tile O_r into one `v16uint8` to perform the MAC operations, as reflected in the accumulator declaration (Line 12).

A single loop iteration retrieves 128+128 elements from memory levels that are close to the arithmetic units, performing $2 \cdot 8 \cdot 8 \cdot 16 = 2,048$ UINT8 arithmetic operations with them. This helps to amortize the cost of memory transfers with enough arithmetic computation, in principle yielding a compute-bound micro-kernel. The high utilization of the accumulator and vector registers (respectively, 100% and 75% of the total resources), along with the appropriate selection of compiler optimization arguments, facilitate to overlap the MAC operations with data transfers.

After the loop is complete, the code "synchronizes" the results with the contents of matrix O in global memory. For that purpose, each execution of the micro-kernel loads a 8×8 micro-tile O_r (Line 46) from global memory and, after updating its contents, stores the results back to global memory (Lines 52). The cost of these data transfers can be amortized provided $c_{i,b}$ is sufficiently large.

4 Experimental Results

The evaluation of the direct algorithm for the CONV operator next was carried out using the Xilinx Vitis 2022.1 developing tool. The AIE transaction-level System C simulator was used to profile the timing, resource requirements, and assembly instructions of the designs, enabling an accurate performance analysis [15].

4.1 Building blocks

As a starting point for the experimental analysis, we focus on the main components that dictate the performance of the blocked realization of the CONV operator. For that purpose, we first evaluate the micro-kernel in a *stationary* scenario

```
#define conv_mac16(v1,v2,v3, zoffsets) \
   v1 = mac16(v1, v2, 0, xoffsets, 16, \
              xsquare, v3, 0, zoffsets, 2, zsquare);

void micro_kernel( input_window_int16 * __restrict DDR_IN,
                   input_window_int16 * __restrict FPGA_IN,
                   uint8 *Fr,
                   output_window_int16 *__restrict DDR_OUT){

  // Vectors for cols of Ir, rows of Fr, and accumulators for Or
  v64uint8 ir0, ir1;
  v32uint8 fr;
  v16acc48 Oacc01, Oacc23, Oacc45, Oacc67;

  // Parameters for mac16() intrinsics
  // Initialization omitted for brevity
  unsigned int xoffsets, xsquare, zoffsets_0, zoffsets_1, zsquare;

  for (unsigned int pr=0; pr<cib; pr+=16)
    // Read 2 x 64 = 64 entries of Ir
    // corresponding to 16 cols with mr=8 UINT8 each
    ir0 = window_readincr_v64(FPGA_IN));
    ir1 = window_readincr_v64(FPGA_IN));

    // Repeat four times:
    //   Read 32 entries of Fr
    //   corresponding to 4 rows with nr=8 UINT each
    //   Use the elements of Ir, Fr to update Or
    fr = *(v32uint8*) Fr[pr/4+0];
    conv_mac(Oacc01, ir0, fr, zoffsets_0);
    conv_mac(Oacc23, ir0, fr, zoffsets_1);

    fr = *(v32uint8*) Fr[pr/4+1];
    conv_mac(Oacc45, ir0, fr, zoffsets_0);
    conv_mac(Oacc67, ir0, fr, zoffsets_1);

    fr = *(v32uint8*) Fr[pr/4+2];
    conv_mac(Oacc01, ir1, fr, zoffsets_0);
    conv_mac(Oacc23, ir1, fr, zoffsets_1);

    fr = *(v32uint8*) Fr[pr/4+3];
    conv_mac(Oacc45, ir1, fr, zoffsets_0);
    conv_mac(Oacc67, ir1, fr, zoffsets_1);
  }
  // Read Or from global memory
  v64uint8 O01234567 = undef_v64uint8();
  O01234567 = window_readincr_v64(DDR_IN);

  // Convert result, add it to Or and write back to memory
  v64uint8 accV01234567 = concat(ubsrs(acc01,0), ubsrs(acc23,0),
                                 ubsrs(acc45,0), ubsrs(acc67,0));
  O01234567 = operator + (accV01234567, O01234567);
  window_writeincr(DDR_OUT, O01234567);
}
```

Fig. 5: Simplified version of the micro-kernel for the AIE tile.

where all data is already "in place". That corresponds to the $m_r \times n_r = 8 \times 8$ micro-tile O_r being retrieved from the global memory; the $m_r \times c_{i,b}$ micro-panel I_r from the local memory; and the $c_{i,b} \times r$ micro-panel F_r from the FPGA RAM. Other memory transfer overheads, in particular the cost of copying I_r from the global memory to the local memory, are omitted from this initial study.

The left plot in Figure 6 displays the throughput rate, in UINT8 MACs/second, attained by the micro-kernel when operating in stationary mode. In this scenario, the $c_{i,b}$ parameter determines the overhead due to loading/storing the micro-tile O_r before/after the execution of loop L6 of the micro-kernel. The plot shows that the micro-kernel attains an asymptotic rate around 110 MACs/cycle, close to the theoretical peak of the AIE tile (128 MACs/cycle).

The blocked direct algorithm for CONV operator requires that, at each iteration of loop L5, a micro-panel I_r is copied from global to local memory, and re-utilized there for all iterations of loop L4; see Figure 3. The right plot in Figure 6 compares the cost of this memory transfer versus that of the micro-kernel execution as a function of $c_{i,b}$. While the right plot gives the initial impression that the memory access costs dominate the execution time of the CONV operator, note that the data transfer overhead is amortized over $c_{o,b}/n_r$ iterations of loop L4. We will expose the actual effect of this memory access overhead when considering real convolution layers in the following subsection.

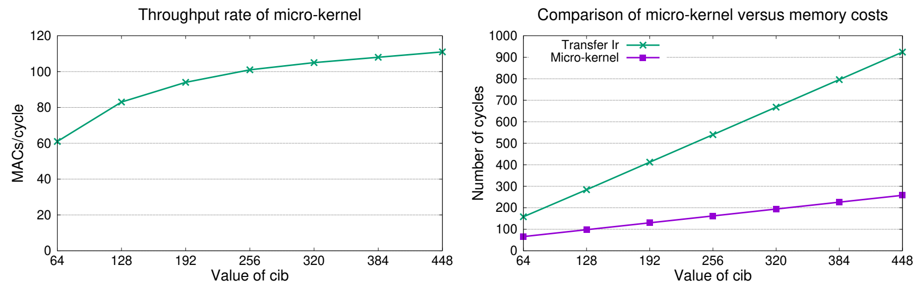

Fig. 6: Impact of the $c_{i,b}$ parameter. Left: Throughput rate of the micro-kernel operating in stationary mode. Right: Cost of micro-kernel versus overhead due to copying I_r from global to local memory.

4.2 Convolutional layers

For the evaluation of the complete convolution algorithm, we selected the convolutional layers in the ResNet-50 v1.5 model. As some of these layers feature the same dimensions, we show only the different ones, listed in Table 1. For the dataset, we chose ImageNet and we set the batch size $b = 1$ (single input scenario). Furthermore, we set the blocking parameters to $c_{o,b} = w_{o,b} = 8192$, and $c_{i,b} = 256$. Note that, in practice, $c_{o,b} \le c_o$, $w_{o,b} \le w_o$, and $c_{i,b} \le c_i$.

Layer id.	c_o	w_o	h_o	w_f	h_f	c_i
C1	64	112	112	7	7	3
C2	256	56	56	1	1	64
C3	64	56	56	1	1	64
C4	64	56	56	3	3	64
C5	64	56	56	1	1	256
C6	512	28	28	1	1	256
C7	128	28	28	1	1	256
C8	128	28	28	3	3	128
C9	512	28	28	1	1	128
C10	128	28	28	1	1	512
C11	1024	14	14	1	1	512
C12	256	14	14	1	1	512
C13	256	14	14	3	3	256
C14	1024	14	14	1	1	256
C15	256	14	14	1	1	1024
C16	2048	7	7	1	1	1024
C17	512	7	7	1	1	1024
C18	512	7	7	3	3	512
C19	2048	7	7	1	1	512
C20	512	7	7	1	1	2048

Table 1: Parameters of the convolutional layers in ResNet-50 v1.5.

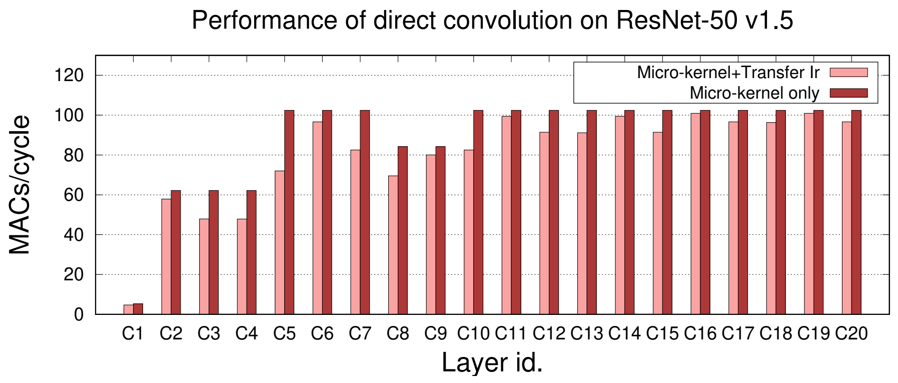

Fig. 7: Performance of the GEMM-like direct algorithm for the CONV operator applied to the distinct convolutional layers of ResNet-50 v1.5+ImageNet.

Given the large variations between the dimension parameters dimensions of the different convolutional layers in ResNet-50 v1.5, in order to evaluate the performance we report the UINT8 arithmetic rate, comparing that with the theoretical peak of the AIE tile. Figure 7 reports the results for this experiment, with the arithmetic throughput rates calculated taking into account the micro-kernel execution in isolation (stationary mode) as well as that cost plus the overhead due to the data transfers.

The results in Figure 7 illustrate that, except for the first few layers (C1–C3), the performance of the direct convolution algorithm implemented on the Xilinx AIE attains a throughput rate superior to 60 MACs/cycle when including the data movement overhead. Furthermore, for a significant number of layers the performance is around 90 MACs/cycles (70% of peak). The low performance arithmetic rate observed for some of the layers can be tracked down to two main causes:

- The number of input channels (c_i, see Table 1) is below 256. This is the case, for example, of layers C1–C4 and C8–C9. The reason is that a low value of c_i constraints $c_{i,b}$, as $c_{i,b} \leq c_i$, and results in a micro-kernel with a lower throughput rate, even in stationary mode; see Figure 6, left. Nonetheless, for those cases with small c_i but large c_o, this problem can be tackled by interchanging the roles of c_i and c_o in the direct algorithm.
- There is a significant overhead due to memory accesses. While not explicitly reported, we can estimate this factor by comparing the cost of the execution with and without considering the data transfers. The overhead is as large as 30% for three of the cases (layers C5, C7 and C8), but in general remains below 12%.

5 Concluding Remarks

We have demonstrated that the same techniques which render high performance from GEMM on a conventional processor, carry over to the AIE tile on the Xilinx Versal. In particular, we refer to an architecture-specific implementation of the micro-kernel, which can be developed using AIE intrinsics, plus a cache-aware layout of the matrix operands and a careful ordering of the nested loops in the convolution operator. The experimental analysis using the convolutional layers in Resnet-50 v1.5+ImageNet shows performance rates which are around 70% of the AIE tile's theoretical peak in UINT8 arithmetic. The analysis exposes that the main limiting factor is the presence of convolution layers with a "small" number of both input and output channels.

Acknowledgments

This project has received funding from the European High-Performance Computing Joint Undertaking (JU) under grant agreement No 955558. The JU receives support from the European Union's Horizon 2020 research and innovation programme, and Spain, Germany, France, Italy, Poland, Switzerland, Norway.

The authors acknowledge funding from European Union's Horizon2020 Research and Innovation programme under the Marie Skłodowska Curie Grant Agreement No. 956090 (APROPOS).

This work was also supported by the research project PID2020-113656RB-C22 of MCIN/AEI/10.13039/501100011033. H. Martínez is a postdoc fellow supported by the *Junta de Andalucía.*

References

1. Ahmad, S., et al.: Xilinx first 7nm device: Versal AI Core (VC1902). In: 2019 IEEE Hot Chips 31 Symposium (HCS). pp. 1–28 (2019)
2. Barrachina, S., et al.: Efficient and portable GEMM-based convolution operators for deep neural network training on multicore processors. J. Parallel and Distributed Computing **167**, 240–254 (2022)

3. Barrachina, S., et al.: Reformulating the direct convolution for high-performance deep learning inference on ARM processors. J. Systems Arch. **135**, 102806 (2023)
4. Ben-Nun, T., Hoefler, T.: Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. ACM Computing Surveys **52**(4), 65:1–65:43 (2019)
5. Castelló, A., Quintana-Ortí, E.S., Igual, F.D.: Anatomy of the BLIS family of algorithms for matrix multiplication. In: 30th Euromicro Int. Conf. on Parallel, Distributed and Network-based Processing (PDP). pp. 92–99 (2022)
6. Chellapilla, K., Puri, S., Simard, P.: High performance convolutional neural networks for document processing. In: International Workshop on Frontiers in Handwriting Recognition (2006), https://hal.inria.fr/inria-00112631
7. Dolz, M.F., et al.: Efficient and portable Winograd convolutions for multi-core processors. The Journal of Supercomputing (2023), to appear
8. Dongarra, J.J., Du Croz, J., Hammarling, S., Duff, I.: A set of level 3 basic linear algebra subprograms. ACM Trans. Math. Softw. **16**(1), 1–17 (March 1990)
9. Goto, K., van de Geijn, R.A.: Anatomy of a high-performance matrix multiplication. ACM Trans. Math. Softw. **34**(3), 12:1–12:25 (May 2008)
10. Lavin, A., Gray, S.: Fast algorithms for convolutional neural networks. In: 2016 IEEE Conf. Computer Vision and Pattern Recognition. pp. 4013–4021 (2016)
11. Low, T.M., et al.: Analytical modeling is enough for high-performance BLIS. ACM Trans. Math. Softw. **43**(2), 12:1–12:18 (Aug 2016)
12. Najafabadi, M.M., et al.: Deep learning applications and challenges in big data analytics. Journal of Big Data **2**(1), 1 (Feb 2015)
13. Sze, V., et al.: Efficient processing of deep neural networks: A tutorial and survey. Proceedings of the IEEE **105**(12), 2295–2329 (Dec 2017)
14. Van Zee, F.G., van de Geijn, R.A.: BLIS: A framework for rapidly instantiating BLAS functionality. ACM Trans. Math. Softw. **41**(3), 14:1–14:33 (2015)
15. Xilinx: AI Engine tools and flows user guide (UG1079). https://docs.xilinx.com/r/en-US/ug1079-ai-engine-kernel-coding/Tools (2022)
16. Zhang, J., Franchetti, F., Low, T.M.: High performance zero-memory overhead direct convolutions. In: Proc. 35th Int. Conf. Machine Learning. vol. 80 (2018)
17. Zhao, Y., et al.: A faster algorithm for reducing the computational complexity of convolutional neural networks. Algorithms **11**(10), 159 (Oct 2018)