Communication-Avoiding Fusion of GEMM-Based Convolutions for Deep Learning in the RISC-V GAP8 MCU

Cristian Ramírez, Adrián Castelló, *Member, IEEE*, Héctor Martínez, and Enrique S. Quintana-Ortí

***Abstract*—Incorporating deep learning (DL) technologies to the edge is crucial for improving the security, privacy, and energy efficiency of the Internet of Things (IoT). In this scenario, the limitations of edge devices in terms of power dissipation, memory capacity, and processing power require a careful selection and optimization of algorithms for IoT DL applications. In this line, our work focuses on the convolution operator, a key component in deep neural networks for signal processing and computer vision. Specifically, the work aims at the efficient implementation of the lowering-based implementation of this operator, on the GAP8 parallel ultralow power platform (PULP), with the goal of mitigating the data transfer costs across the memory hierarchy. Our contributions include 1) an analytical model for estimating the parallel execution time, 2) the exploration of different configuration options, and 3) four variants of the algorithm that fuse several components to address memory bottlenecks in the method. Overall, our best fused variant provides a speedup of up to 1.25× over the baseline algorithm when applied to infer MobileNet-v1+ImageNet and VGG9+CIFAR10 using eight threads, and up to 1.34× for ResNet18+ImageNet.**

***Index Terms*—Convolution, convolutional neural networks (CNNs), cost analysis, deep learning (DL), edge processors, high performance.**

I. Introduction

DEPLOYING Internet of Things (IoT) appliances, many of which use deep learning (DL) technologies, is critical to improving security and privacy, reducing latency for end users, and/or diminishing energy consumption [1], [2], [3]. However, in many cases, edge processors have severe limitations in terms of power supply, memory capacity, and/or processing power. As a result, it is crucial to carefully select and optimize the algorithms for IoT DL-based applications.

The convolution is a key operator for the type of deep neural networks (DNNs) that are nowadays applied with great success in signal processing and computer vision tasks [4], [5]. In response, a large effort has been spent over the past years to optimize this type of computational kernel on a large variety of architectures, from multicore processors from Intel, AMD, ARM, and IBM, to many-core accelerators, such as AMD/NVIDIA's graphics processing units (GPUs) as well as Google's tensor processing units (TPUs) [6], [7], [8].

The convolution is often realized via the *lowering approach*, which transforms the input tensor into an augmented matrix, casting the operator as a general matrix-matrix multiplication (GEMM) [9], [10], [11]. This GEMM-based approach is highly flexible and delivers fair performance on virtually any conventional processor due to the existence of optimized, architecture-specific instances of GEMM.

With this general objective in mind, in this article we target the efficient realization of the lowering-based convolution operator on the heterogeneous 1+8 RISC-V cores integrated into the GAP8 parallel ultralow power platform (PULP) [12]. Our work makes the following specific contributions.

1) We propose an analytical model that accurately estimates the execution time of the parallel implementation of the convolution operator, via the lowering approach. This analytical model is then leveraged to expose that, on the PULP platform, the performance of the parallel implementation is strongly limited by the data transfers across the memory hierarchy.
2) In order to tackle the memory bottleneck, we propose several communication-avoiding enhancements to the baseline algorithm for the lowering approach that considerably reduce the cost of data movements.

The remainder of this article is structured as follows. In Section II we offer a brief review of the convolution operator and its basic implementation via the lowering approach. In Section III, we describe a high performance, parallel implementation of the GEMM-based convolution for the GAP8 PULP. In Section IV we introduce a performance model to estimate the arithmetic and data transfer costs of the parallel GEMM-based algorithm for the convolution. In Section V we introduce four variants of the baseline convolution algorithm with distinct levels of memory requirements and data transfers. Finally, in Section VI we summarize the main insights from this analysis.

II. Convolution via Lowering

The convolution operator

$$O = \text{CONV}(F, I) \quad (1)$$

Manuscript received 5 June 2024; accepted 29 July 2024. Date of publication 1 August 2024; date of current version 24 October 2024. This work was supported in part by MCIN/AEI/10.13039/501100011033 under Project PID2020-113656RB and Project TED2021-129334B-I00, and in part by "EU NextGenerationEU/PRTR." The work of Héctor Martínez was supported by the Junta de Andalucía (POSTDOC_21_00025). *(Corresponding author: Adrián Castelló.)*

Cristian Ramírez, Adrián Castelló, and Enrique S. Quintana-Ortí are with the DISCA, Universitat Politècnica de València, 46022 Valencia, Spain (e-mail: crirabe@posgrado.upv.es; adcastel@disca.upv.es; quintana@disca.upv.es).

Héctor Martínez is with the Electronic and Computer Engineering, Universidad de Córdoba, 14071 Córdoba, Spain (e-mail: el2mapeh@uco.es).

Digital Object Identifier 10.1109/JIOT.2024.3436937

© 2024 The Authors. This work is licensed under a Creative Commons Attribution 4.0 License.
For more information, see https://creativecommons.org/licenses/by/4.0/

```
1  for (i=0; i<b; i++)
2   for (j=0; j<ci; j++)
3    for (k=0; k<wo; k++)
4     for (l=0; l<ho; l++) {
5      r = l + k + i*wi*hi;
6      for (m=0; m<wf; m++)
7       for (n=0; n<hf; n++) {
8        c = m + wf*n + j*wf*hf;
9        A[r][c] = I[i][l + n][k][j];
10    } }
```

Fig. 1. Algorithm for the IM2ROW transform.

applies a 4D (4-D) filter tensor F of size $c_i \times h_f \times w_f \times c_o$, to a 4D input tensor I of size $b \times h_i \times w_i \times c_i$, producing a 4D output tensor O of size $b \times h_o \times w_o \times c_o$. In these dimensions, b corresponds to the number of (input and output) images (also known as batch size); each input|output image, of size $h_i \times w_i | h_o \times w_o$ (height × width), consists of c_i input|c_o output channels; and there are $c_i c_o$ filters, each of size $h_f \times w_f$ (height × width). For simplicity, we assume that the filter is applied with unit vertical/horizontal strides; and the output is not padded so that $h_o = h_i - h_f + 1, w_o = w_i - w_f + 1$.

The lowering approach to compute the convolution operator "flattens" the input tensor into an augmented matrix, via the IM2ROW transform [9], casting (1) into a large GEMM

$$C = A \cdot B \tag{2}$$

where $C \equiv O$ is the output tensor, viewed as an $m \times n = (bh_o w_o) \times c_o$ matrix; and $B \equiv F$ is the filter tensor, viewed as a $k \times n = (c_i h_f w_f) \times c_o$ matrix. Furthermore, the *augmented matrix* A, of size $m \times k = (bh_o w_o) \times (c_i h_f w_f)$, results from applying the IM2ROW transform to the input tensor I as shown in the algorithm in Fig. 1; see also [9]. In practice, for the IM2ROW transform, the input and output tensors are stored in memory following the NHWC layout, where N specifies the batch dimension; $H \times W$ refer to the image height × width; and C corresponds to the channels.[1]

III. Two-Stage Lowering on Edge Processor—The GAP8

The lowering approach decomposes the convolution operator into two "stages," corresponding to the IM2ROW transform followed by a "large" GEMM. The first stage basically involves data copies and movements to assemble the $m \times k$ augmented matrix A. Here, we note that this may impose severe requirements on the memory capacity of the system as the dimension of A is $h_f \cdot w_f$ times larger than that of the input tensor I. The matrix multiplication in the second stage is a well-known computational kernel that, when operating on large matrices, can be executed efficiently on current architectures with a multilayered memory, via a blocked algorithm. In this section, we first review the main characteristics of the target GAP8 PULP. Then, we describe how to adapt the blocked algorithm for GEMM to attain high performance on this edge platform.

[1]The alternative IM2COL also yields a GEMM, in this case flattening the input tensor into an augmented matrix B involved in the matrix product. The IM2ROW transform is ideal for tensors stored in the NCHW format.

A. Target Platform

In our work, we select the GAP8 as representative of a state-of-the-art low-power platform for IoT applications. This system is based on the PULP architecture and, as depicted in Fig. 2, embeds three main computing components: 1) a low-power micro-controller unit (MCU), known as the FC, responsible for managing control, communications, and security functions; 2) a compute engine (CE) comprising a cluster of 8 compute cores specifically for the execution of parallel algorithms; and 3) a specialized hardware accelerator (HWCE) that is part of the CE as well.

The FC contains a read-only memory (ROM) and a private 16KB S1 scratchpad (or Memory Area, MA). On the CE side, the compute cores and HWCE share a 64-KB S1 scratchpad. In addition, the FC and CE share a 512-KB S2 scratchpad. The device also contains an 8-MB S3 (or M), which acts as the platform's main memory and is accessible from the FC. To enable fast data transfer between memory areas, the platform has two direct memory access (DMA) units.

Both the FC and cluster cores support the RISC-V *RV32IMCXpulpV2* instruction set architecture (ISA), including instructions for the vector *dot product* (DOT) [13].

B. Algorithmic Variant of GEMM for the GAP8

Modern, high-performance implementations of GEMM mimic GotoBLAS [14] to formulate this computational kernel as five nested loops embedding two packing routines and a micro-kernel. In these libraries, the micro-kernel is the only architecture-specific code, comprising an additional loop that performs an outer product, which is decomposed into a collection of SAXPY (scalar α times x plus y) operations [13].

The special support for the DOT in the GAP8 PULP led us to choose the A3C2B0[2] member of the family of algorithms for matrix multiplication [15], [16], [17]. This particular variant of GEMM decomposes the operation into a collection of DOT operations inside the micro-kernel, each involving part of two rows from A_c and C_c, respectively, with k_r and n_r elements, and a $k_r \times n_r$ micro-tile of B. The variant is illustrated in Fig. 3, which also shows the data movements across the memory of a platform equipped with a five-level memory system: registers, three levels of cache, and main memory; and the packing of the matrix operands [18]. For simplicity, hereafter, we assume that m, n, k are, respectively, integer multiples of m_c, n_c, k_c; and n_c, k_c are, respectively, integer multiples of n_r, k_r.

C. Customization of GEMM for the GAP8 PULP

In [19], the A3C2B0[3] variant of GEMM is adapted for the GAP8 PULP taking into account that 1) the memory hierarchy consists of four levels: vector registers, two intermediate scratchpad levels (referred to as S1, S2), and a main memory

[2]In this notation, $Z \in$ {A,B,C} specifies one of the three matrix operands and the subsequent number, $i \in \{0, 2, 3\}$, indicates the cache level where that operand resides (with 0 referring to the processor registers). The same matrix resides in both the L1 and L3 caches.

[3]Indeed the algorithmic variant of GEMM described in [19] corresponds to B3C2A0. However, as A, B play symmetric roles in matrix multiplication, there are no major differences between that variant and A3C2B0. We select the latter because it is more natural to combine with the IM2ROW transform.

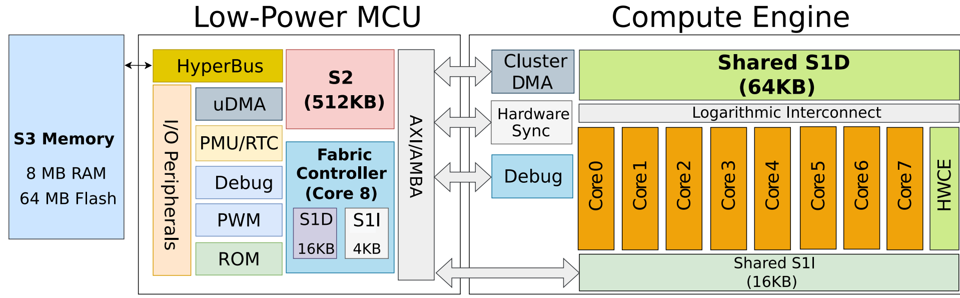


Fig. 2. GAP8 layout.

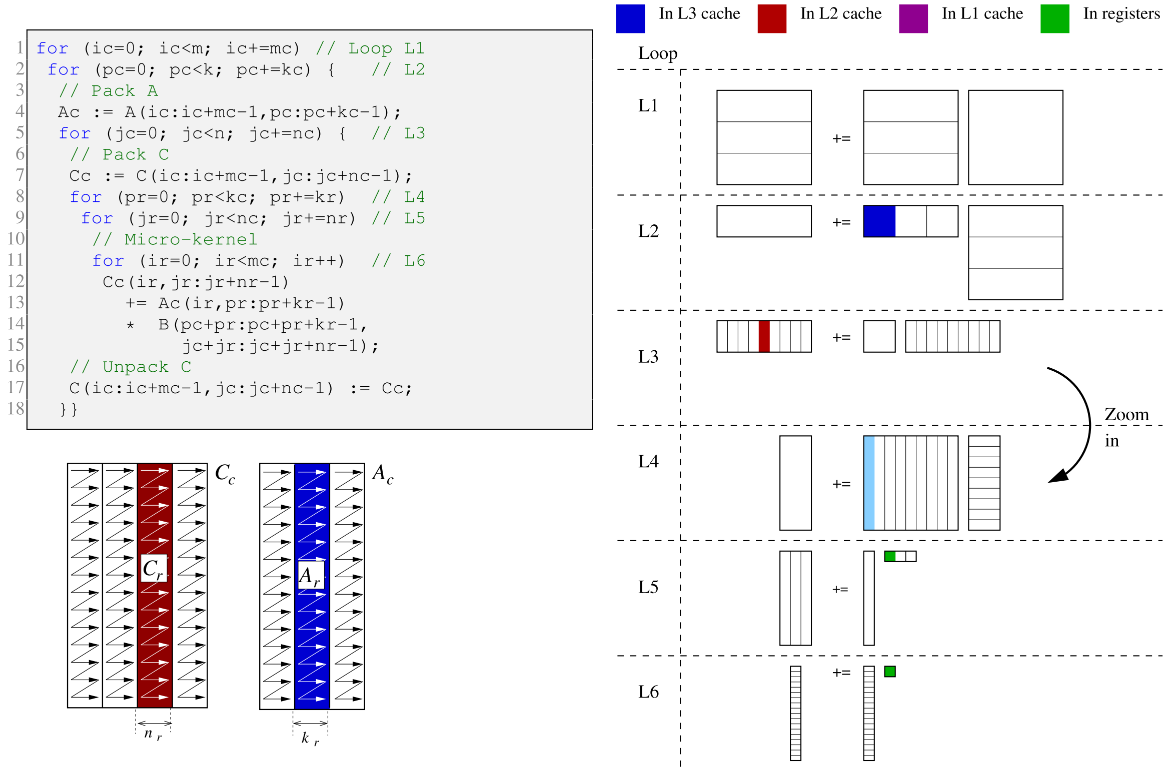


Fig. 3. Top-left: A3C2B0 variant of GEMM for the GAP8 PULP (here, C_c and A_c are buffers that maintain copies of certain blocks of C and A). Bottom-left: Packing of the data into the buffers. Right: Data movements across the memory hierarchy.

area (or M); 2) a single FC controls the memory transfers; and 3) the CE features 8 compute cores. The A3C2B0 operates in parallel with all the compute cores of the CE as follows.

1) MICRO-KERNEL: The FC and compute cores support the same RISC-V-oriented ISA, including the DOT product. Adapting the initial FC micro-kernel from [20] for the CE cores basically required no changes.
2) DATA TRANSFERS: The FC is in charge of all the copies involving the main memory and the S2. Those include the packing of A, C, the unpacking into C and the copying of A_c. In addition, the FC performs the copy of the micro-tiles of B from the main memory to the S1. When executing a micro-kernel, the compute cores access the data for the micro-panel C_r from the S2, for the micro-panel A_r from the S1, and for the micro-tile B_r from the S1. This is illustrated in Fig. 3 (right).
3) PARALLELIZATION: Our scheme extracts loop parallelism by distributing the iteration space of loop L5 in the A3C2B0 variant. As a result, all the compute cores access the same $m_c \times n_c$ buffer C_c in the S2, the same $m_c \times k_r$ micro-panel A_r in the S1, but a different $k_r \times n_r$ micro-tile B_r in the S1; see [19].

The fact that the GAP8 integrates scratchpad memories instead of hardware-assisted caches imposes on the programmer the task of orchestrating the data transfers across the different levels of the memory hierarchy. On the positive side, this enables the possibility of obtaining accurate estimates of the transfer rates via simple bandwidth benchmarks. In the following, we will refer to the memory transfer rates using the notation $R_{\text{So,Sd}}$ (in INT8 numbers per second), where So, Sd $\in$ {R(egisters), S1, S2, M(emory)}, respectively, specify the origin and destination of the transfer. In

TABLE I
SUMMARY OF COSTS (IN S) OF THE TWO-STAGE ALGORITHM FOR THE LOWERING APPROACH

Component	Equation	Cost
T_{arith}	(6)	$2\,m\,k\,n/(R_A\,c)$
T_{streamCr}	(9)	$2\,m\,k\,n\,(1/R_{\mathsf{S2,R}}+1/R_{\mathsf{R,S2}})\,/\,(k_r\,c)$
T_{streamAr}	(11)	$m\,k\,n/(R_{\mathsf{S1,R}}\,n_r\,c)$
T_{streamB}	(14)	$m\,k\,n\,(1/R_{\mathsf{M,S1}}+1/R_{\mathsf{S1,R}})/(m_c\,r_B)$
T_{packA}	(16)	$m\,k/(R_{\mathsf{M,M}}\,k_r)$
T_{packC}	(18)	$m\,k\,n/(R_{\mathsf{M,S2}}\,k_c\,n_r)$
T_{unpackC}	(20)	$m\,k\,n/(R_{\mathsf{S2,M}}\,k_c\,n_r)$
T_{copyAr}	(22)	$m\,k\,n/(R_{\mathsf{M,S1}}\,k_r\,c\,r_A)$
T_{IM2ROW}	(24)	$t_{IR}/R_{\mathsf{M,R}}+t_{IW}/(R_{\mathsf{R,M}}\,r_I)$

addition, we will use R_A to denote the arithmetic rate (in INT8 operations per second).

For the evaluation of the parallel performance of this algorithm on the GAP8 PULP, we refer the reader to [19].

IV. ANALYSIS OF THE TWO-STAGE LOWERING APPROACH

This section analyzes the arithmetic and data movement costs of the two-stage algorithm underlying the lowering approach. The performance model described and validated next is extended from [21] to accommodate a parallel execution on a heterogeneous platform composed of multiple cores that perform the computation, in addition to an FC in charge of most data transfers. The goal of the performance model is to explore the effect of different algorithmic variants, without going through the effort of actually implementing them.

In the following, we consider the IM2ROW and GEMM as two separate, *nonfused stages*, which are executed independently, one after the other. In the costs, the subindices for the summations specify the bounds defined by the corresponding loop in Fig. 3. Thus, for example, the summation $\sum_{\mathtt{L1}}$ refers to loop `L1` there, which iterates from $i_c = 0$ to $m-1$ in steps of m_c. All costs are expressed in s(econds). Finally, we assume a parallel execution using c threads/cores of the CE.

A. Micro-Kernel

All the relevant arithmetic in the lowering approach is performed inside the micro-kernel, which is invoked from inside loop `L5` of the A3C2B0 algorithm a total of

$$\sum_{\mathtt{L1}}\sum_{\mathtt{L2}}\dots\sum_{\mathtt{L5}} 1 = \frac{m}{m_c}\,\frac{k}{k_c}\,\frac{n}{n_c}\,\frac{k_c}{k_r}\,\frac{n_c}{n_r} \quad (3)$$

$$= m\,k\,n/(m_c\,k_r\,n_r) \text{ times.} \quad (4)$$

At each iteration of loop `L6`, the micro-kernel executes a vector-matrix product involving a row of the micro-panel A_r and the $k_r \times n_r$ micro-tile B_r to update a row of the micro-panel C_r. The number of addition/multiplication operations per vector-matrix product is thus $2\,k_r\,n_r$ and, since loop `L6` comprises m_c iterations, the theoretical parallel execution time of GEMM due to the arithmetic is given by

TABLE II
EXPERIMENTAL TRANSFERS RATES IN THE GAP8. NOTE THAT, SINCE WE CONSIDER INT8 AS THE BASIC DATATYPE, THE BYTES/S RATES DIRECTLY TRANSLATE INTO INT8/S

	Mbytes/s	
$R_{\mathsf{M,M}}$	2.85E+00	Pack A to A_c (by FC)
$R_{\mathsf{M,R}}$	1.82E−01	Stream B to reg. (by CE, via S1)
$R_{\mathsf{R,M}}$	1.82E−01	(by CE, via S1)
$R_{\mathsf{M,S2}}$	5.82E−01	Pack C to C_c (by FC)
$R_{\mathsf{S2,M}}$	7.17E−01	Unpack C_c to C (by FC)
$R_{\mathsf{M,S1}}$	7.76E+01	Copy A_c to A_r
$R_{\mathsf{S2,R}}$	1.76E+01	Stream C_r to reg.
$R_{\mathsf{R,S2}}$	1.76E+01	Stream C_r from reg.
$R_{\mathsf{S1,R}}$	4.87E+03	Stream A_r to reg.

$$T_{\text{arith}} = \sum_{\mathtt{L1}}\sum_{\mathtt{L2}}\dots\sum_{\mathtt{L5}} m_c\,(2\,k_r\,n_r)/(R_A\,c) \quad (5)$$

$$= 2\,m\,k\,n/(R_A\,c)\ s. \quad (6)$$

The micro-kernels stream their data between the core registers and three different levels of the memory hierarchy (see Fig. 5 (right)): 1) the rows of the $m_c \times n_r$ micro-panel C_r from S2 and back there once they are updated; 2) the rows of the $m_c \times k_r$ micro-panel A_r from S1; and 3) the $k_r \times n_r$ micro-tile B_r from S1 (after being copied there from the main memory).

In summary, the transfer costs in 1)–3), due to the execution of the micro-kernel, are, respectively, given by

$$T_{\text{streamCr}} = \sum_{\mathtt{L1}}\sum_{\mathtt{L2}}\dots\sum_{\mathtt{L5}} m_c\,n_r \quad (7)$$

$$\left(1/R_{\mathsf{S2,R}} + 1/R_{\mathsf{R,S2}}\right)/\,c \quad (8)$$

$$= 2\,m\,k\,n\left(1/R_{\mathsf{S2,R}} + 1/R_{\mathsf{R,S2}}\right)/\,(k_r\,c)\ s \quad (9)$$

$$T_{\text{streamAr}} = \sum_{\mathtt{L1}}\sum_{\mathtt{L2}}\dots\sum_{\mathtt{L5}} m_c\,k_r/\left(R_{\mathsf{S1,R}}\,c\right) \quad (10)$$

$$= m\,k\,n/\left(R_{\mathsf{S1,R}}\,n_r\ c\right)\ s \quad (11)$$

$$T_{\text{streamB}} = \sum_{\mathtt{L1}}\sum_{\mathtt{L2}}\dots\sum_{\mathtt{L5}} k_r\,n_r \quad (12)$$

$$\left(1/R_{\mathsf{M,S1}} + 1/R_{\mathsf{S1,R}}\right) \quad (13)$$

$$= m\,k\,n\left(1/R_{\mathsf{M,S1}} + 1/R_{\mathsf{S1,R}}\right)/m_c\ s. \quad (14)$$

Now, when accessing data that is stored in blocks (or "chunks") of r consecutive elements, we experimentally determined that, for small values of r, the transfers from the main memory are accelerated by a factor of r with respect to the reference transfer rates in the expressions. Taking into consideration this experimental observation, in case we prepack the entries of B into blocks of $k_r \cdot n_r$ consecutive elements in memory, each micro-tile B_r can be read as a single chunk. Therefore, the transfer cost applied to the streaming of B in (14) is divided by a factor $r_B = k_r \cdot n_r$. Prepacking B is possible because in the IM2ROW variant of the lowering approach, this matrix contains the convolution filters, which remain constant across inference with multiple samples.

We also note that the streaming costs reflect that the streaming of C_r and A_r proceed in parallel while that of B proceeds serially; see the experimental evaluation in [19].

TABLE III
PARAMETERS OF THE CONVOLUTION LAYERS ARISING IN MOBILENET-V1 (TOP), VGG9 (MIDDLE), AND RESNET18 (BOTTOM), AND DIMENSIONS OF THE GEMM OBTAINED WITH THE APPLICATION OF THE IM2ROW TRANSFORM

	Layer identifier (id.)	Convolution parameters						GEMM dimensions		
		c_o	w_o	h_o	h_f	w_f	c_i	m	n	k
MobileNet-v1	1	32	224	224	3	3	3	50,176	32	27
	2	32	112	112	3	3	32	12,544	32	288
	3	64	112	112	1	1	32	12,544	64	32
	4	64	56	56	3	3	64	3,136	64	576
	5	128	56	56	1	1	128	3,136	128	128
	6	128	56	56	3	3	128	3,136	128	1,152
	7	128	56	56	1	1	128	3,136	128	128
	8	128	28	28	3	3	128	784	128	1,152
	9	256	28	28	1	1	128	784	256	128
	10	256	28	28	3	3	256	784	256	2,304
	11	256	28	28	1	1	256	784	256	256
	12	256	14	14	3	3	256	196	256	2,304
	13	512	14	14	1	1	256	196	512	256
	14,16,18,20,22	512	14	14	3	3	512	196	512	4,608
	15,17,19,21,23	512	14	14	1	1	512	196	512	512
	24	512	7	7	3	3	512	49	512	4,608
	25	1,024	7	7	1	1	512	49	1024	512
	26	1,024	7	7	3	3	1,024	49	1024	9,216
	27	1,024	7	7	1	1	1,024	49	1024	1,024
VGG9	1	32	32	32	3	3	3	1,024	32	27
	2	64	16	16	3	3	32	256	64	288
	3	128	16	16	3	3	64	256	128	576
	4	128	16	16	3	3	128	256	128	1,152
	5	256	8	8	3	3	128	64	256	1,152
	6	256	8	8	3	3	256	64	256	2,304
ResNet18	1	64	112	112	7	7	3	12,544	64	147
	2,3,4,5	64	56	56	3	3	64	3,136	64	576
	6	128	28	28	3	3	64	784	128	576
	7,8,9	128	28	28	3	3	128	784	128	1,152
	10	256	14	14	3	3	128	196	256	1,152
	11,12,13	256	14	14	3	3	256	196	256	2,304
	14	512	7	7	3	3	256	49	512	2,304
	15,16,17	512	7	7	3	3	512	49	512	4,608

B. Data Transfers Outside the Micro-Kernel

The A3C2B0 algorithm for GEMM performs a number of data copies external to the micro-kernel, which are discussed next to estimate their cost. For this purpose, we assume hereafter that A, C are stored in row-major order.

In the A3C2B0 algorithm, an $m_c \times k_c$ block of A is packed into the buffer A_c inside the two outermost loops. This involves a copy from memory to memory (via the FC registers), yielding the following cost due to these packings:

$$T_{\text{packA}} = \sum_{\texttt{L1}} \sum_{\texttt{L2}} m_c\, k_c / R_{\textsf{M,M}} \tag{15}$$
$$= m\, k / R_{\textsf{M,M}}\ s. \tag{16}$$

Now, since A is stored row-wise and A_c has to be assembled in micro-panels of width k_r, these copies can be done in "chunks" of k_r consecutive elements in memory and, in such case, the transfer cost has to be divided by this factor.

For the transfers associated with data movements between C in memory and the $m_c \times n_c$ buffer C_c in $\textsf{S2}$, inside the three outermost loops of the A3C2B0 algorithm, we have a total cost

$$T_{\text{packC}} = \sum_{\texttt{L1}} \sum_{\texttt{L2}} \sum_{\texttt{L3}} m_c\, n_c / R_{\textsf{M,S2}} \tag{17}$$
$$= m\, k\, n / \big(R_{\textsf{M,S2}}\, k_c\big)\ s \tag{18}$$

and

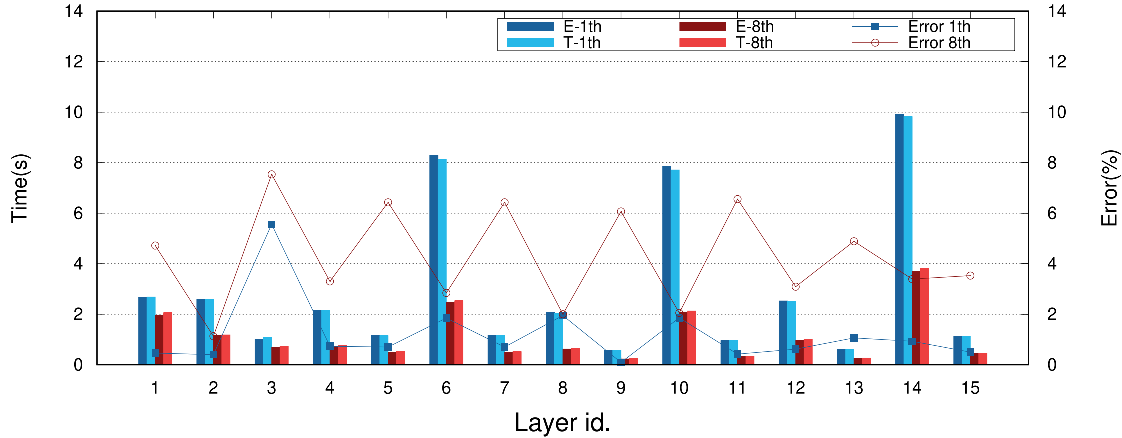
$$T_{\text{unpackC}} = \sum_{\texttt{L1}} \sum_{\texttt{L2}} \sum_{\texttt{L3}} m_c\, n_c / R_{\textsf{S2,M}} \tag{19}$$
$$= m\, k\, n / \big(R_{\textsf{S2,M}}\, k_c\big)\ s. \tag{20}$$

Since C is stored row-wise and C_c is organized in micro-panels of width n_r, the data movements can be done, in both cases, in chunks of n_r elements, and the corresponding transfer costs need to be divided by the factor n_r.

Following the inspection of A3C2B0 from the outermost to the innermost loops, we next identify the copy of an $m_c \times k_r$ micro-panel of A_c in memory to A_r in $\textsf{S1}$, with a cost

$$T_{\text{copyAr}} = \sum_{\texttt{L1}} \sum_{\texttt{L2}} \sum_{\texttt{L3}} \sum_{\texttt{L4}} m_c\, k_r / \big(R_{\textsf{M,S1}}\, c\big) \tag{21}$$
$$= m\, k\, n / \big(R_{\textsf{M,S1}}\, k_r\, c\big)\ s. \tag{22}$$

In this case the micro-panel resides contiguously in memory after the packing and, therefore, the transfer can be done in a single chunk of $m_c \cdot k_r$ elements. Now, in contrast with the previous cases, this value can be large. In consequence, we assume a transfer cost that is divided by a factor $r_A = \min(\max_r, m_c \cdot k_r)$, where $\max_r$ denotes the maximum acceleration that can be attained by performing the data transfers in chunks. (For simplicity, in our cost expressions we use



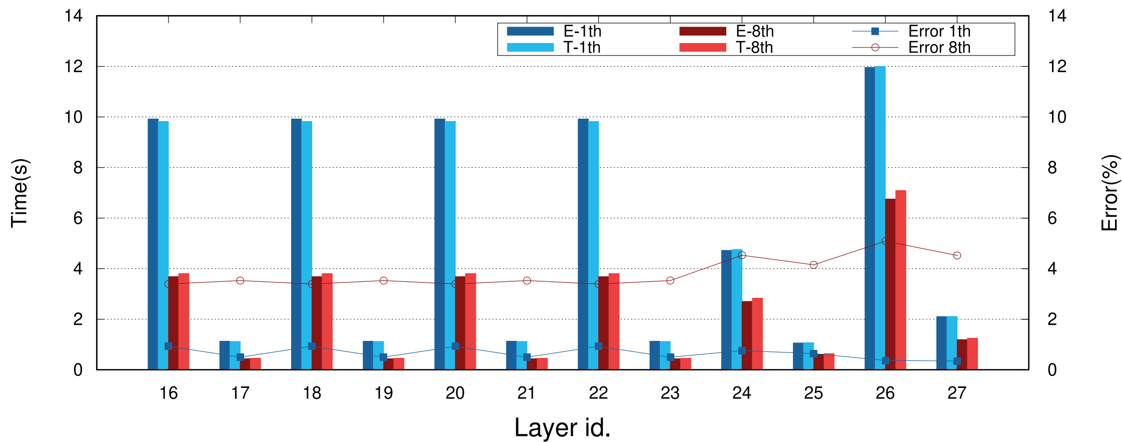


Fig. 4. Comparison of the experimental execution time versus performance model using 1 and 8 cores of the GAP8 PULP.

$\max_r$ to denote the maximum speed-up independently of the source/destination of the transfer.)

The packings in this section involve the FC, and they all proceed serially, while the copy of A_r, runs in parallel; see the experimental evaluation in [19].

C. Data Transfers in IM2ROW

The copies of the IM2ROW algorithm incur the cost

$$T_{\text{IM2ROW}} = b\, c_i\, w_o\, h_o \big(1/R_{\mathsf{M,R}} + w_f\, h_f / R_{\mathsf{R,M}}\big) \quad (23)$$
$$= t_{IR}/R_{\mathsf{M,R}} + t_{IW}/R_{\mathsf{R,M}}\ s \quad (24)$$

as each entry of the input tensor I is replicated into $w_f \cdot h_f$ entries of the augmented matrix A. Now, in case we reorder the loops of the IM2ROW algorithm so that the innermost one traverses the w_f dimension, we can read a chunk of w_f entries of I from memory one by one, but they can then be written back to memory as a single chunk. In such case, the transfer cost associated with the writes has to be divided by $r_I = \min(\max_r, w_f)$. Furthermore, for a filter with $w_f \times h_f = 1 \times 1$, we can then reorganize the loops of the IM2ROW algorithm so that we can write a chunk of c_i consecutive elements to memory, yielding $r_I = \min(\max_r, c_i)$.

Table I compiles the costs derived in this section as a function of the convolution input/output parameters. Remember that $m = bh_ow_o$, $k = c_ih_fw_f$, and $n = c_o$. In these expressions, some of the transfer rates correspond to copies between the memory and the registers/memory of the FC. For example, this is the case of the cost for packing A into A_c or for the IM2ROW transform. In the table, we highlight in blue the acceleration factors due to blockwise accesses to the data. Table II displays the transfer rates determined experimentally for the GAP8 platform.

In [22], we introduced a performance model for the execution of the convolution on the FC of the GAP8 PULP. The performance model presented in this section and validated next mainly differs from that one in that it estimates the costs for a parallel execution, using the FC *plus the eight compute cores*.

D. Validation

For the evaluation of the performance model, we employ the MobileNet-v1 DNN with the ImageNet data set, setting the batch size $b = 1$, as corresponds to a single input scenario. Table III (top) lists the dimension parameters of the convolution layers in this model. MobileNet-v1 includes a fully connected layer (id. 28), but this boils down to a

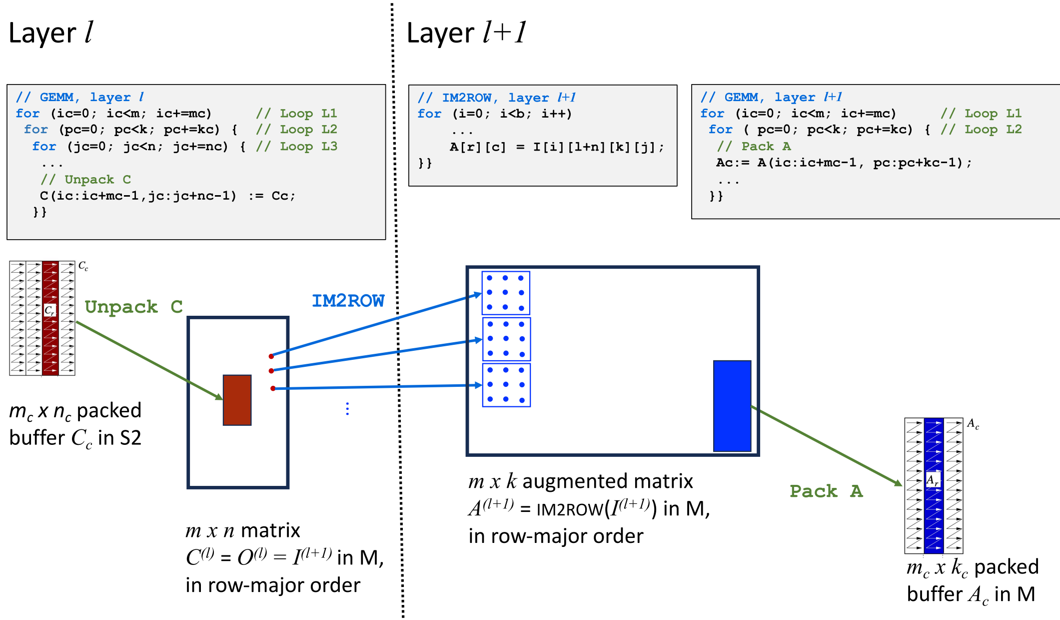


Fig. 5. Connection between the output of GEMM for layer l and the input of lowering (IM2ROW+GEMM) for layer $l+1$ in the baseline, nonfused algorithm for the lowering approach. The convolution output $O^{(l)} = C^{(l)}$ for layer l is a matrix of dimension $(b \cdot h_o^{(l)} \cdot w_o^{(l)}) \times c_o^{(l)}$. The IM2ROW transform applied in the layer $l+1$ to $I^{(l+1)} = C^{(l)}$, produces the augmented matrix $A^{(l+1)}$, of dimension $(b \cdot h_o^{(l+1)} \cdot w_o^{(l+1)}) \times (c_i^{(l+1)} \cdot h_f^{(l+1)} \cdot w_f^{(l+1)})$.

TABLE IV

COSTS (IN S) OF THE FUSED CONFIGURATIONS FOR LOWERING APPROACH. WHENEVER THE COST EQUALS THAT OF THE BASELINE (TWO-STAGE) ALGORITHM, THE CORRESPONDING ENTRY OF THE TABLE IS LEFT EMPTY. THE COSTS OF THE ARITHMETIC, STREAMING THE DATA FROM THE MICRO-KERNEL, AND THE COPY OF A_r DO NOT VARY WITH RESPECT TO THE BASELINE ALGORITHM

Component	IMPA	IMPA_OTF	UCIM	FULL
T_{packA}	0	$t_{IR}/R_{\text{M,R}} + t_{IW}/(R_{\text{R,M}}\, k_r)$		
T_{packC}				$m\, k\, n/(R_{\text{M,S2}}\, k'_c\, r_C)$
T_{unpackC}			$m\, k\, n/(R_{\text{S2,M}}\, k'_c\, n_r) + t_{IW}/(R_{\text{S2,M}}\, r_I)$	$m\, k\, n/(R_{\text{S2,M}}\, k'_c\, r_C) + t_{IW}/(R_{\text{S2,M}}\, r_I)$
T_{IM2ROW}		0	0	0

matrix-vector product and, therefore, is omitted from the experiments. For all layers, the vertical/horizontal paddings equal 1.

Fig. 4 compares the execution times of sequential and parallel executions of the baseline algorithm for the convolution with the costs estimated with the performance model described earlier. The results in these two plots show that the model model accurately estimates the experimental execution time in the sequential case, with an error margin that is below 2% in all cases except for layer id. 3, where it is still below 6%. For the parallel model, the deviations with respect to the experimental time are quite moderate, below 8% in all cases and around 4% only on average. The main message from this evaluation is that the performance model offers a reasonable tool to predict the execution time of the parallel convolution algorithm on the GAP8 PULP.

V. FUSED CONFIGURATIONS FOR THE LOWERING APPROACH

This section illustrates the effect on performance of fusing the IM2ROW transform with some of the packing/unpacking operations inside GEMM in order to reduce the volume of data transfers across the memory hierarchy.

A convolutional neural network (CNN) is composed of several convolution layers connected by elementwise transforms, such as nonlinear functions, and in some cases, pooling, batch normalization, etc. Since these other types of layers can often be merged with the previous convolution, for brevity we will omit them from the following discussion. We use superscripts in the tensor/matrix operands where necessary to distinguish the layer. Thus, for example, $A^{(l)}$ and $A^{(l+1)}$ refer to the augmented matrices assembled in layers l and $l+1$, respectively. Note that the output of the convolution operation in layer l (possibly after some minor transformations, which we omit for simplicity) becomes the input tensor for $l+1$; that is, $C^{(l)} = O^{(l)} = I^{(l+1)}$.

A. Baseline Convolution

The baseline, nonfused algorithm executes the two stages of the lowering approach layer-by-layer, first applying the IM2ROW followed by the GEMM of each layer; see Fig. 5. During the execution:

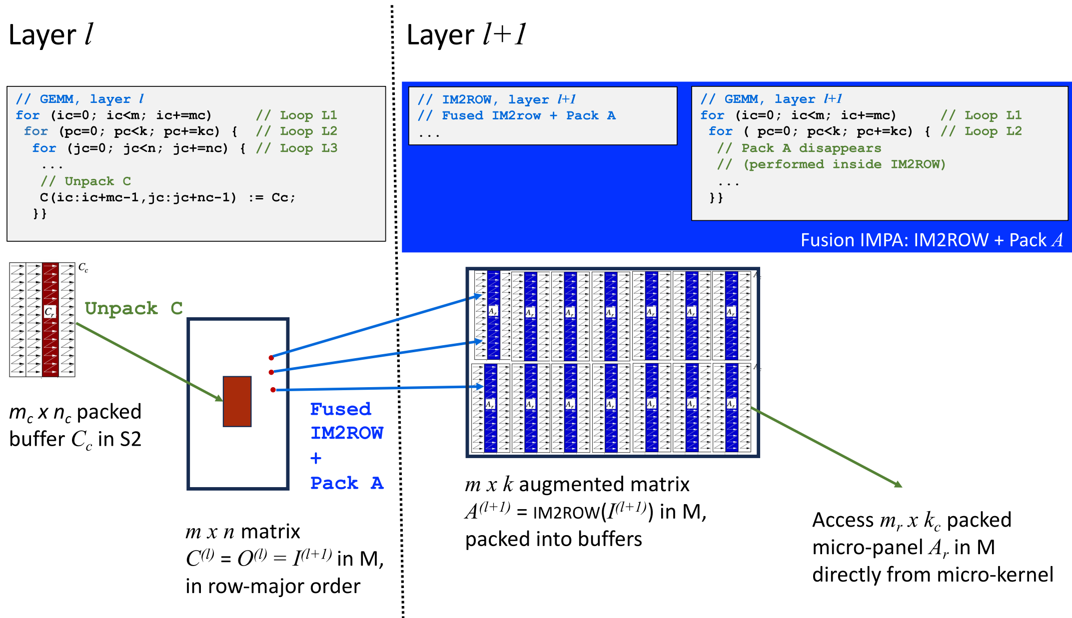


Fig. 6. Fusion IMPA: IM2ROW + Pack A.

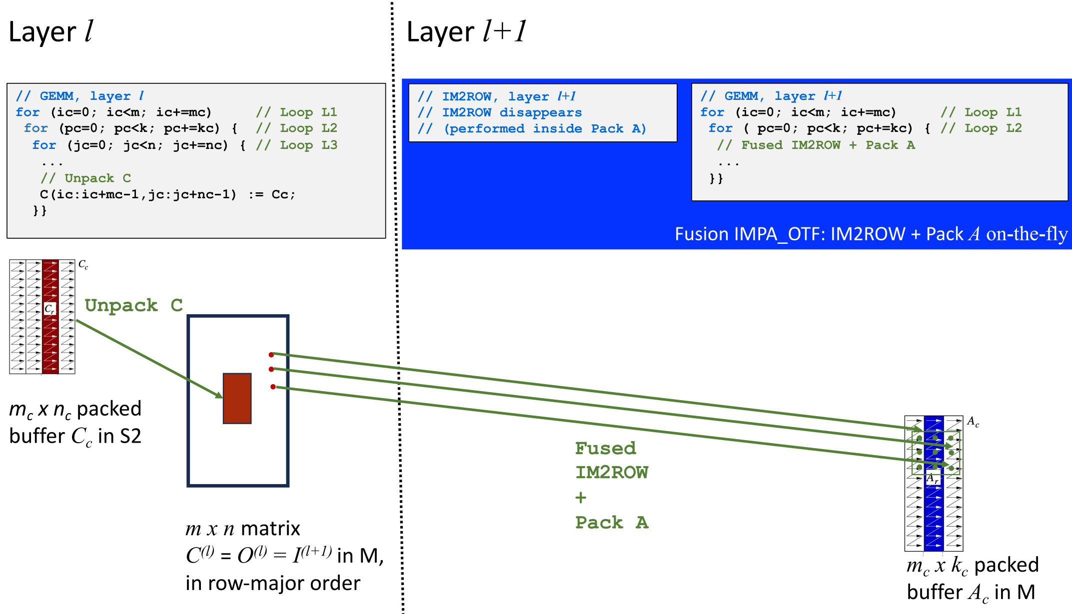


Fig. 7. Fusion IMPA_OTF: IM2ROW + Pack A "OTF."

1) The unpacking operations in loop L3 of the GEMM in layer l copy the entries of the buffer C_c from S2 to the appropriate locations in the row-major order matrix $C^{(l)}$ that resides in M.
2) Once the execution of the GEMM in layer l is completed, the algorithm applies the IM2ROW transform in layer $l+1$, involving a reorganization and replication of the data of $C^{(l)} = I^{(l+1)}$ into $A^{(l+1)}$, from M to M. This replicates the elements of the input tensor $I^{(l+1)}$ into the appropriate entries of the augmented matrix $A^{(l+1)}$, to be feed as a one of the matrix operands for the GEMM in the subsequent layer $(l+1)$.
3) During the matrix multiplication in layer $l+1$, the blocks of the augmented matrix $A^{(l+1)}$, residing in M in row-major order, are packed into the buffer A_c (loop L2), also in M.

Separately, the IM2ROW and GEMM components of a layer (among others) incur the costs for the packing of A, C, the unpacking of C, and the IM2ROW, respectively, given in Table I. We aim to reduce or eliminate these transfer costs via fusion. Concretely, we present four fused solutions, discussed in the following sections: The first two configurations combine part of the data movements inside the same layer l, concretely, IM2ROW + Pack A. The second configuration merges the

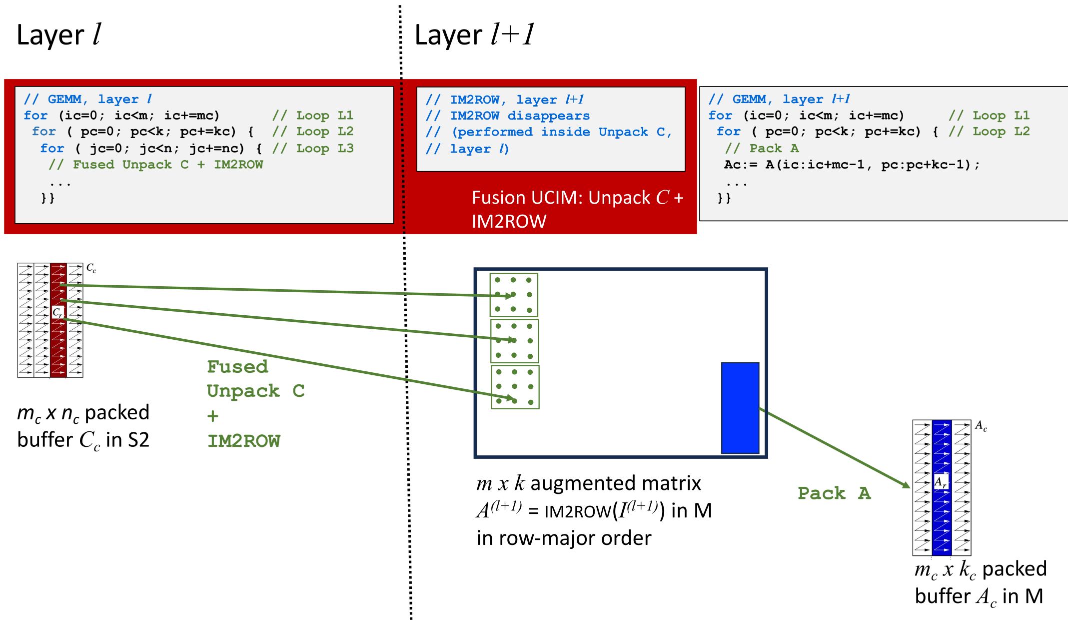


Fig. 8. Fusion UCIM: Unpack C + IM2ROW. Last update of the buffer C_c.

unpacking of the GEMM in layer l with the IM2ROW transform of layer $l+1$: Unpack C + IM2ROW. Finally, in the fully fused configuration, we combine the three operations: Unpack C + IM2ROW + Pack A.

B. Fusion IMPA—IM2ROW + Pack A

In the fused configuration IMPA, we apply the IM2ROW in stage $l+1$ but, instead of writing the result to memory into $A^{(l+1)}$ as a plain matrix in row-major order, we store its entries already in "packed mode." In other words, as part of IM2ROW, the $m \times k$ augmented matrix $A^{(l+1)}$ is already laid out as a collection of packed buffers; see Fig. 6.

With this approach, the cost of applying the IM2ROW transform does not vary: in the innermost loop of the IM2ROW algorithm, we read elements from $I^{(l+1)}$ in M one-by-one and these are then written back into M in chunks of r_I elements. However, the writes are now done into the appropriate entries of the $m \times k$ array $A^{(l+1)}$ in M, taking into account the packing scheme that the subsequent GEMM expects. Thus, since the elements of $A^{(l+1)}$ are already packed, at each iteration of loop L2 in the GEMM in layer $l+1$, there is no need to rearrange a block of this matrix into the buffer A_c.

In summary, with this type of fusion, the cost of packing $A^{(l+1)}$ into A_c disappears while the cost of the IM2ROW transform remains the same; see Table IV.

C. Fusion IMPA_OTF—IM2ROW + Pack A "on-the-Fly"

In [11], we proposed an alternative for the lowering-based convolution that modifies the packing of the B3A2C0 algorithm GEMM to apply the IM2COL transform on the input tensor I "on-the-fly" (OTF), that is, as part of the packing of this operand during the execution of the matrix multiplication. As a result, we avoid the explicit assembly of the full augmented matrix, requiring no extra workspace other than the buffer A_c that is already employed for GEMM.

The technique introduced in [11] carries over to the A3C2B0 algorithm. The idea is similar to that already described in Section V-B, with the difference that the IM2ROW is performed from inside the packing routine for $A^{(l+1)}$ whenever this is invoked during the execution of the GEMM in layer $l+1$; see Fig. 7.

With this scheme, the cost of applying the IM2ROW transform in (24) is merged with that of packing $A^{(l+1)}$. As part of this packing, we have to read $m_c \times k_c$ elements of $I^{(l+1)}$ from M, in principle one-by-one because they reside in noncontiguous positions, to store them into the appropriate entries of the buffer A_c in M. These data copies can be arranged so that the writes are performed in chunks of k_r elements. We can, therefore, estimate the cost of the fused IM2ROW + packing A by

$$t_{IR}/R_{\mathsf{M,R}} + m\,k/\big(R_{\mathsf{R,M}}\,k_r\big) \tag{25}$$
$$= t_{IR}/R_{\mathsf{M,R}} + t_{IW}/\big(R_{\mathsf{R,M}}\,k_r\big) \tag{26}$$

where the first addends in the sum reflect that the reads are done elementwise, while the second ones consider that the writes are done in chunks of k_r elements.

The costs of this type of fusion are displayed in the corresponding column in Table IV. Compared with the baseline algorithm, the cost of the IM2ROW transform disappears as it is merged with that of packing A. The latter's cost basically matches that of the IM2ROW transform in the baseline algorithm, with the difference that the writes are now done in chunks of k_r elements instead of r_I.

D. Fusion UCIM—Unpack C + IM2ROW

Instead of combining the IM2ROW transform (stage 1) in layer l with the packing of $A^{(l)}$ inside the GEMM (stage 2) in

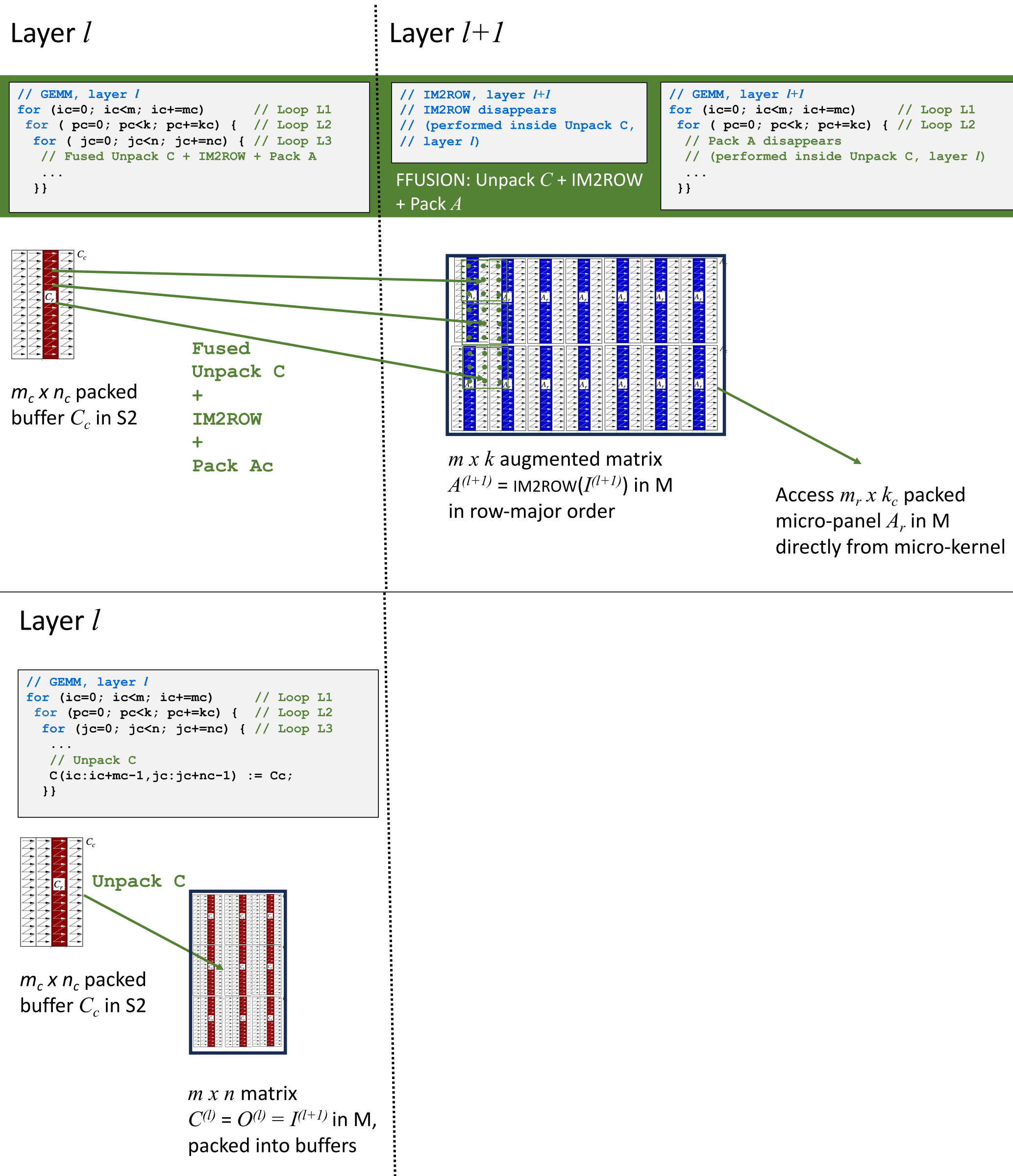


Fig. 9. Fusion FULL: Unpack C + IM2ROW + pack A. Top: Last update of the buffer C_c. Bottom: Other updates.

the same layer, we can fuse the stores onto C during the unpacking of the buffer C_c inside the GEMM (stage 2) of layer l, with the IM2ROW transform (stage 1) to be applied in the next layer $l+1$. That is, during the unpacking of C_c in layer l, we assemble the augmented matrix for the next layer: $A^{(l+1)}$; see Fig. 8.

With this scheme, we have to copy the entries of the $m_c \times n_c$ buffer C_c into the appropriate positions of the augmented matrix $A^{(l+1)}$, but *this has to be done only after the last update of each buffer* C_c. For the remaining updates, we unpack the data in C_c into the output matrix for this layer into $C^{(l)}$ in row-major order. In consequence, the cost of the IM2ROW transform in (24) is merged with that of the unpacking, which becomes

$$T_{\text{unpackC}} = \sum_{L1} \sum_{L2'} \sum_{L3} m_c\, n_c / \left(R_{\mathsf{S2,M}}\, n_r\right) \quad (27)$$

$$+\, t_{IW} / \left(R_{\mathsf{S2,M}}\, r_I\right) \quad (28)$$

$$= m\, k\, n / \left(R_{\mathsf{S2,M}}\, k_c\, n_r\right) + t_{IW} / \left(R_{\mathsf{S2,M}}\, r_I\right). \quad (29)$$

In the first addend of this expression, $\sum_{L2'}$ is a loop that performs $k'_c = (k/k_c) - 1$ iterations, corresponding to those steps where the output of the unpacking is written onto $C^{(l)}$. The second addend corresponds to the cost of storing the entries of all the $m_c \times n_c$ buffers C_c from S2 (which globally are equivalent to the full $m \times n$ matrix $C^{(l)}$) as the augmented matrix $A^{(l+1)}$ in M. There we assume that C_c can be copied from S2 written to M in chunks of r_I elements at a time.

With this type of fusion, the cost of the IM2ROW is shifted to the last time each buffer C_c is unpacked, being performed then from S2 to M, instead of from M to M as was the case in the baseline algorithm; see Table IV.

From the storage point of view, this solution implies that we need to maintain two augmented matrices in memory simultaneously: the matrix for the current layer $A^{(l)}$, which is being used during the GEMM of the current layer, and the matrix for the next layer $A^{(l+1)}$, which is being built at the same time.

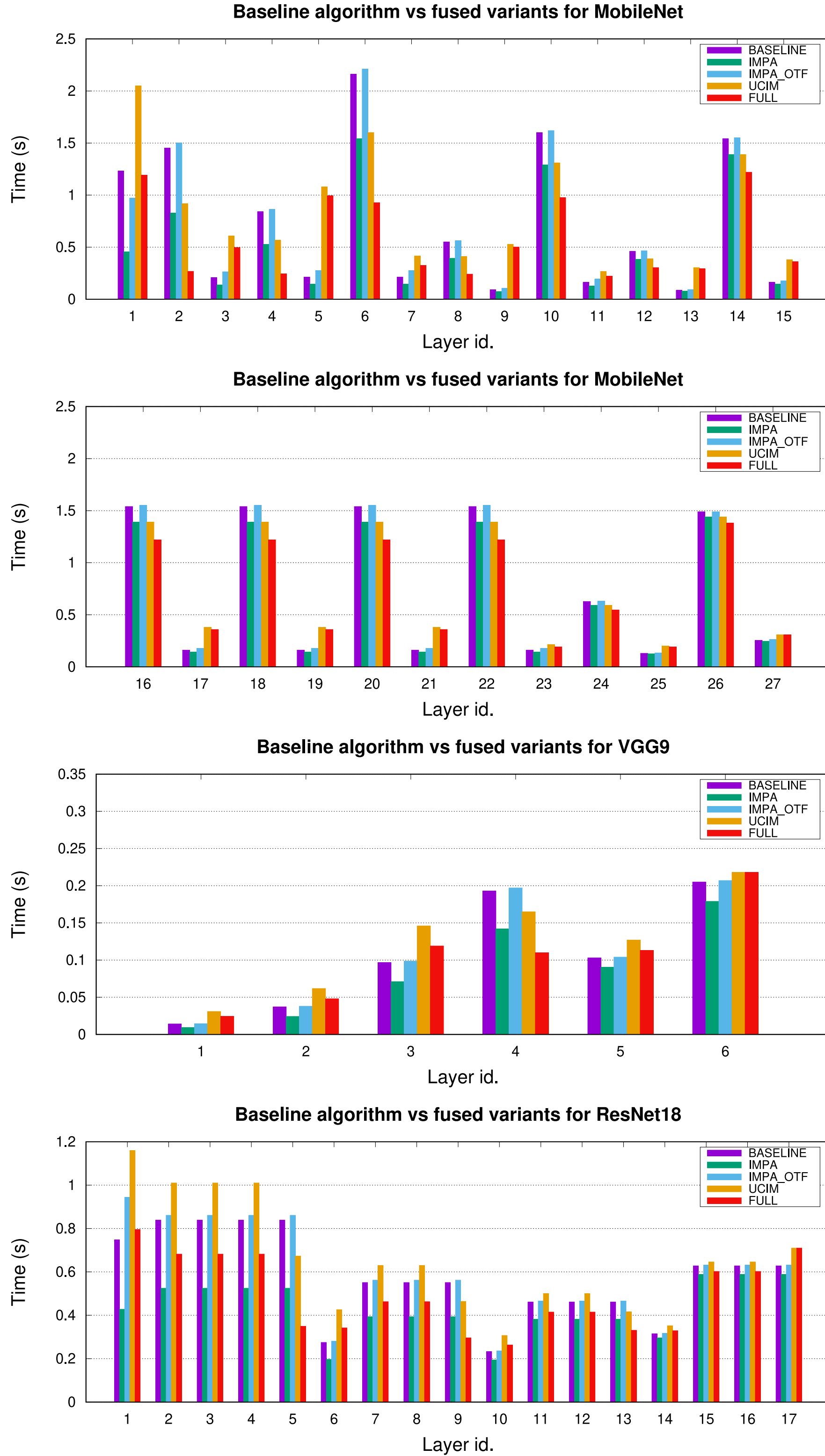


Fig. 10. Execution time of the baseline algorithm for the convolution and the four fused variants, using eight CE cores, for MobileNet-v1 (top two plots), VGG9 (third plots), and ResNet18 (bottom plot).

E. Fusion FULL—Unpack C + IM2ROW + Pack A

The final solution enhances the fusion scheme UCIM so that, right after the last update of the buffer C_c, we simultaneously unpack it, apply the IM2ROW transform, and store the result into the augmented matrix $A^{(l+1)}$ packed as a collection of buffers A_c; see Fig. 9 (top). In addition, for all iterations except the one that performs the last update on the buffer C_c, instead of writing the entries of this buffer into the matrix $C^{(l)}$

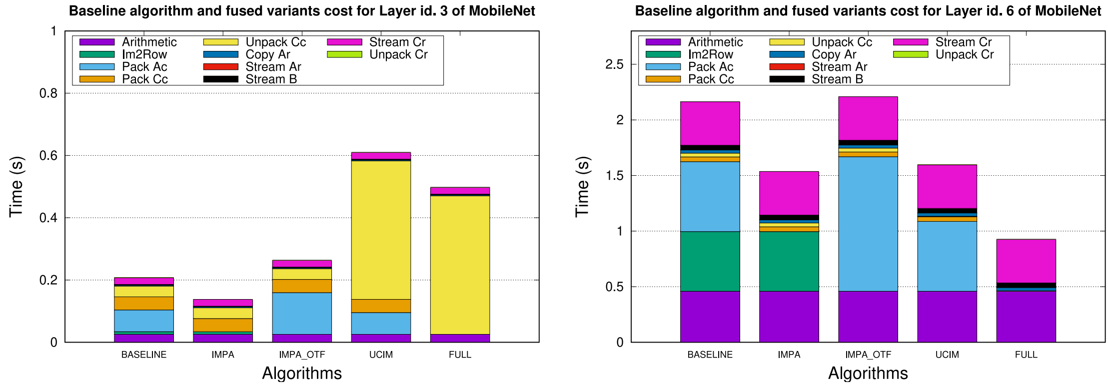


Fig. 11. Execution time of the baseline algorithm for the convolution and the four fused variants, using eight CE cores, for layer 3 (left) and layer 6 (right) of MobileNet-v1+ImageNet.

TABLE V
MEMORY REQUIREMENTS OF BASELINE

Level	Operand	Dimensions	Comment
R	B_r	$m_r \cdot n_r$	Micro-tile for GEMM
S1	A_r	$m_r \cdot k_c$	Micro-panel for GEMM
S2	C_c	$m_c \cdot n_c$	Buffer for GEMM
M	A_c	$m_c \cdot k_c$	Buffer for GEMM
M	$O \mid C$	$(b \cdot h_o \cdot w_o) \times c_o = m \times n$	Output tensor of CONV | Output of GEMM
M	I	$(b \cdot h_i \cdot w_i) \times c_i$	Input tensor of CONV
M	A	$(b \cdot h_i \cdot w_i) \times (c_i \cdot h_f \cdot w_f) = m \times k$	Augmented matrix, input of GEMM
M	$F \mid B$	$(c_i \cdot h_f \cdot w_f) \times c_o = k \times n$	Filter tensor of CONV | Input of GEMM

in row-major order, we store them as a collection of packed buffers; see Fig. 9 (bottom).

With this final fusion scheme, the cost of packing A disappears. Furthermore, the entries of the buffer C_c can be moved between S2 and M in large chunks both when unpacking them and packing them. The resulting costs are listed in Table IV where we take into account that the transfer cost for packing/unpacking C is reduced by a factor $r_C = \max_r$. (Indeed, the packing behaves much like a copy of the block, as there is no need to rearrange the data.) We also exploit that, since the GEMM that is being multiplied is $C = A \cdot B$, the first access to each macro-tile C_c only requires initializing the entries of this block to zero. Therefore, the number of packing operations is reduced to k_c'.

F. Performance Analysis

We now compare the estimated costs of the baseline algorithm against those of the four fused variants for sequential and parallel executions on 1, 2, 4, and 8 threads/CE cores (plus the FC) of the GAP8 PULP, using three DNN models: MobileNet-v1+ImageNet, VGG9+CIFAR10, and ResNet18+ImageNet. Before doing so, we remind that UCIM and FULL both shift the cost of the IM2ROW in layer $l + 1$ into the unpacking operations of the previous layer l. For this reason, a direct comparison of their performance with the remaining convolution options for an individual layer is difficult. However, we can still compare these two variants against each other for individual layers; furthermore, we can elaborate a separate comparison for individual layers of the trio BASELINE-IMPA-IMPA_OTF.

Fig. 10 displays the costs of a parallel execution of the baseline algorithm and the four fused variants, IMPA, IMPA_OTF, UCIM, and FULL, using eight cores. The global trend shows that FULL is clearly superior to UCIM; while IMPA outperforms BASELINE as well as IMPA_OTF. An additional trend we can observe in those plots is that the differences between the options are larger for the first layers but decrease as we move toward the final ones.

In order to expose and, consequently, understand better the sources of the differences, Fig. 11 breaks down the execution times into the distinct components for two specific layers of MobileNet-v1 that we identified as representative of the general behavior. Comparing the trio BASELINE-IMPA-IMPA_OTF, we observe in both layers/plots that the superior performance of IMPA is due to the elimination of the costs due to packing A (light blue bar). The source of the differences in favor of FULL compared with UCIM is mostly the same plus, in the specific case of layer 3, the reduction of the cost of packing C (orange bar). While there is a similar reduction of the cost for packing C for layer 6, that particular component is not relevant in that case and, therefore, the reduction effect is almost invisible.

From the plots in Fig. 11 we also observe that there are significant variations of the cost sources depending on the layer. Thus, for some layers, the contribution of the arithmetic is minor compared to other dominating components. For example, this is the case for layer 3 and the pair UCIM-FULL. In some other cases, the arithmetic, the streamings, and the copies from A_c onto A_r are significant. At this point, it is important to remark that our fused variants only target the cost for the IM2ROW and packing/unpacking. Therefore,

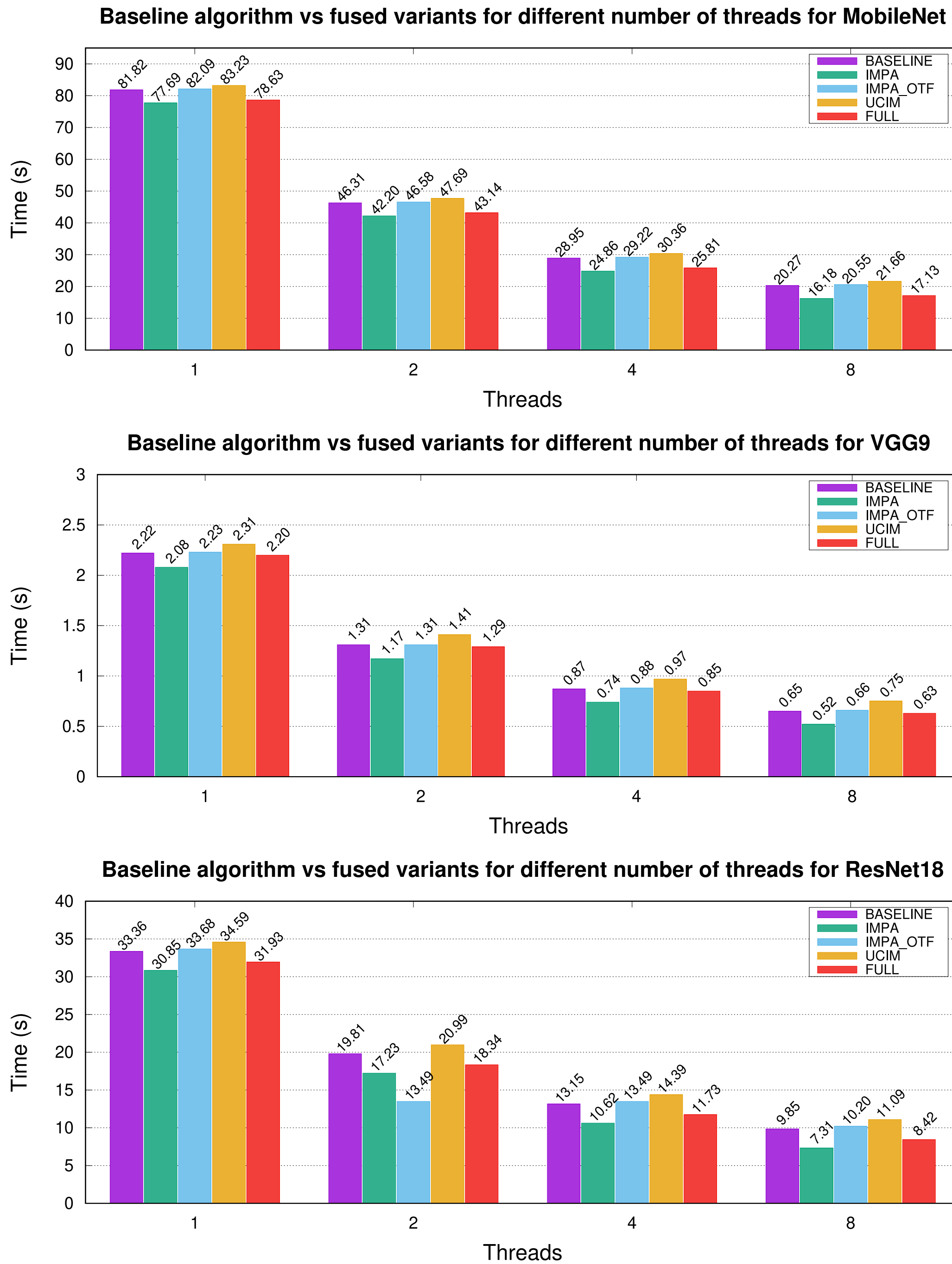


Fig. 12. Execution time of the baseline algorithm for the convolution and the four fused variants, using 1, 2, 4, and 8 CE cores, for MobileNet-v1 (top plot), VGG9 (middle plot), and ResNet18 (bottom plot).

the expected global acceleration effect will be limited by the fraction that these components represent over the total cost.

Fig. 12 shows the global comparison of the execution times for all the convolution options. As could be expected, IMPA and FULL are the best options. In addition, the former is slightly superior to the latter, independently of the number of cores. A close inspection of the cost components revealed that the reason for this is the faster transfer rate $R_{\mathsf{R,M}}$ when this copy is performed by the FC during the IM2ROW transform, compared with the slower transfer rate $R_{\mathsf{S2,M}}$ that applies when a CE core does the equivalent of the IM2ROW transform, fused into the unpacking of C. This is not compensated by the elimination of the packing costs of A in FULL.

In global, IMPA delivers acceleration factors with respect to BASELINE that grow with the number of cores: between 1.05× with 1 core and 1.25× with 8 cores for both MobileNet-v1 and VGG9; and 1.08× with 1 core and 1.34× with 8 cores for ResNet18.

G. Memory Requirements

The fused variants vary the memory transfer costs with respect to the baseline case (see Table IV), but also differ in the memory requirements. In particular, Table V specifies the memory consumption of BASELINE. Compared with that, both IMPA and FULL eliminate the need for the buffer A_c in M, because the micro-kernel accesses these data in the augmented matrix A. The memory costs of IMPA_OTF are even more reduced because it assembles the augmented matrix on the fly directly into the buffer A_c, and thus eliminates the need for the large augmented matrix A in M. Finally, UCIM presents the same memory requirements as BASELINE.

VI. Conclusion

Our study underscores the critical role of addressing the memory bottleneck in edge processors for IoT DL applications by carefully optimizing the implementation of the convolution operator. In this line, we propose an analytical model for accurately estimating the execution time on the 1+8-core GAP8 platform, revealing limitations imposed by memory transfers. In response, we introduce communication-avoiding variants to mitigate this constraint, enhancing the overall efficiency.

The present analysis employs a quantization scheme that assumes that both input operands to the convolution operator are consigned as INT8 matrices, and the accumulations (dot products) that are necessary to obtain each entry of the result are performed using INT8 arithmetic. Unless the operations are carefully monitored, this approach may lead to overflow. To avoid this, a more practical approach would perform the multiplication in INT8 arithmetic and the accumulation on INT32 (or floating-point), followed by a quantization of the results. However, this requires a very special support for mixed precision arithmetic from the hardware in the GAP8 PULP. Furthermore, we expect that such change would not vary the main message of the work, which is the reduced memory transfer costs achieved by the fused variants.

Our findings indicate that the exploration is ongoing, suggesting a compelling avenue for hardware-software co-design (architecture-algorithm). Specifically, in response to the identified hardware bottlenecks, one could tailor the memory bandwidth of specific levels by integrating faster DMA engines and, at the same time, embedding quicker arithmetic units. On the software front, alleviating hardware constraints could involve integrating a double buffering mechanism for certain packing operations. In summary, this work presents a foundation for future advancements in enhancing the synergy between hardware and software components in IoT DL applications.

References

[1] K. Hazelwood et al., "Applied machine learning at Facebook: A datacenter infrastructure perspective," in *Proc. IEEE Int. Symp. HPC Archit.*, 2018, pp. 620–629.

[2] J. Park et al., "Deep learning inference in Facebook data centers: Characterization, performance optimizations and hardware implications," 2018, *arXiv:1811.09886*.

[3] C. Wu et al., "Machine learning at Facebook: Understanding inference at the edge," in *Proc. IEEE Int. Symp. HPC Archit.*, 2019, pp. 331–344.

[4] T. Ben-Nun and T. Hoefler, "Demystifying parallel and distributed deep learning: An in-depth concurrency analysis," *ACM Comput. Surveys*, vol. 52, no. 4, p. 65, Aug. 2019.

[5] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proc. IEEE*, vol. 105, no. 12, pp. 2295–2329,Dec. 2017.

[6] "Intel OneAPI deep neural network library." 2023. [Online]. Available: https://www.intel.com/content/www/us/en/developer/tools/oneapi/onednn.html

[7] "NVIDIA cuDNN developer guide." 2023, [Online]. Available: https://docs.nvidia.com/deeplearning/cudnn/developer-guide/index.html

[8] N. P. Jouppi et al., "In-datacenter performance analysis of a tensor processing unit," *SIGARCH Comp. Arch. News*, vol. 45, no. 2, pp. 1–12, 2017.

[9] K. Chellapilla, S. Puri, and P. Simard, "High performance convolutional neural networks for document processing," in *Proc. 10th Int. Workshop Front. Handwriting Recogn.*, 2006, pp. 1–6.

[10] E. Georganas et al., "Anatomy of high-performance deep learning convolutions on SIMD architectures," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage, Anal.*, 2018, pp. 1–12.

[11] P. San Juan, A. Castelló, M. F. Dolz, P. Alonso-Jordá, and E. S. Quintana-Ortí, "High performance and portable convolution operators for multicore processors," in *Proc. IEEE 32nd Int. Symp. Comput. Arch. High Perform. Comput. (SBAC-PAD)*, 2020, pp. 91–98.

[12] A. Pullini et al., "Mr. Wolf: An energy-precision scalable parallel ultra low power SoC for IoT edge processing," *IEEE J. Solid-State Circuits*, vol. 54, no. 7, pp. 1970–1981, Jul. 2019.

[13] C. L. Lawson et al., "Basic linear algebra subprograms for fortran usage," *ACM Trans. Math. Soft.*, vol. 5, no. 3, pp. 308–323, Sep. 1979.

[14] K. Goto and R. A. van de Geijn, "Anatomy of a high-performance matrix multiplication," *ACM Trans. Math. Softw.*, vol. 34, p. 12, 2008.

[15] T. M. Smith and R. A. van de Geijn, "The MOMMS family of matrix multiplication algorithms," 2019, *arXiv:1904.05717*.

[16] J. A. Gunnels, F. G. Gustavson, G. M. Henry, and R. A. van de Geijn, "A family of high-performance matrix multiplication algorithms," in *Proc. 7th Int. Conf. Appl. Parallel Comput. State Art Sci. Comput.*, 2004, pp. 256–265.

[17] A. Castelló, E. S. Quintana-Ortí, and F. D. Igual, "Anatomy of the BLIS family of algorithms for matrix multiplication," in *Proc. 30th Euromicro Int. Conf. Parallel, Distrib. Netw.-Based Process. (PDP)*, 2022, pp. 92–99.

[18] F. G. Van Zee and R. A. van de Geijn, "BLIS: A framework for rapidly instantiating BLAS functionality," *ACM Trans. Math. Softw.*, vol. 41, no. 3, p. 14, 2015.

[19] C. Ramírez, A. Castelló, H. Martínez, and E. S. Quintana-Ortí, "Parallel GEMM-based convolution for deep learning on multicore RISC-V processors," *J. Supercomputing*, vol. 80, pp. 12623–12643, Feb. 2024. [Online]. Available: https://www.researchsquare.com/article/rs-3394564/v1

[20] C. Ramírez, A. Castelló, and E. S. Quintana-Ortí, "A BLIS-like matrix multiplication for machine learning in the RISC-V ISA-based GAP8 processor," *J. Supercomput.*, vol. 78, pp. 18051–18060, May 2022.

[21] C. Ramírez, A. Castelló, H. Martínez, and E. S. Quintana-Ortí, "Performance analysis of matrix multiplication for deep learning on the edge," in *Proc. ISC Int. Workshops*, 2023, pp. 65–76.

[22] P. Alonso-Jordá, H. Martínez, E. S. Quintana-Ortí, and C. Ramírez, "Performance analysis of convolution algorithms for deep learning on edge processors," in *Parallel Processing and Applied Mathematics* (Lecture Notes in Computer Science 13827), R. Wyrzykowski, J. Dongarra, E. Deelman, and K. Karczewski, Eds., Cham, Switzerland: Springer, 2023, pp. 236–247. [Online]. Available: https://doi.org/10.1007/978-3-031-30445-3_20