Horizon 2020 Program (2014-2020)
FET-Open – Novel ideas for radically new technologies
FETOPEN-01-2018-2019-2020

Architecting More than Moore – Wireless Plasticity for Massive Heterogeneous Computer Architectures [†]

D4.2: In-Memory Accelerator

Contractual Date of Delivery	31/03/2021
Actual Date of Delivery	30/03/2021
Deliverable Dissemination Level	Public
Editor	Davide Rossi (UNIBO)
Contributors	UNIBO (leader), IBM
Quality Assurance	Sergi Abadal (UPC)

[†]This project is supported by the European Commission under the Horizon 2020 Program with Grant agreement no: 863337.

Document Revisions & Quality Assurance

Deliverable Number	D4.2
Deliverable Responsible	UNIBO
Work Package	WP4
Main Editor	Davide Rossi

Internal Reviewers

1. Alexandre Levisse (EPFL)
2. Mohamed Elsayed (RWTH)

Revisions

Version	Date	By	Overview
1.0.0	04/03/2021	*Davide Rossi (UNIBO*	First draft
1.0.1	04/03/2021	*Gianmarco Ottavi (UNIBO)*	Added cluster description
1.0.2	04/03/2021	*Geethan Karunaratne (IBM)*	Added IMA contribution
1.0.3	04/03/2021	*Davide Rossi (UNIBO)*	Finalized draft
1.1.0	30/03/2021	*Davide Rossi (UNIBO), Geethan Karunaratne (IBM)*	Revised version addressing internal review comments

Legal Disclaimer

Executive Summary

The main subject of D4.2 is to describe the analog in-memory computing cores used as building blocks for wirelessly enabled massively parallel heterogeneous architectures explored in WiPlash. It first describes the in-memory computing arrays as well as their main performance metric, correlated with silicon measurements, and the different models used for system-level integration and evaluation. Then it describes the integration of the analog in memory computing cores into a software-programmable cluster of RISC-V processors, highlighting the configurable of the programming interface and the interactions between analog and digital domains needed to accomplish not trivial computing tasks. Finally it presents an exploration of the proposed cluster and its evaluation on the basis of a MobileNetV2 bottleneck benchmark, representative of an emerging amount of modern DNN workloads, highlighting the need for more tightly coupled integration of analog and digital accelerators in next generation computing systems. These findings will be exploited in the second half of the project to develop the target heterogeneous architectures. This deliverable is related to task T4.2: ”In-Memory Computing Accelerator”, M6-M18, led by IBM and T4.3 ”Accelerators Interface”, M6-M18 led by UNIBO. All the activities related to these two tasks have been successfully completed. The output of the activities related to this deliverable will be used as input for developing the whole massively parallel system architecture in T4.1.

Abbreviations and Acronyms

CNN	Convolutional Neural Network
IMA	In-Memory Accelerator
TCDM	Tightly Coupled Data Memory
HWPE	Hardware Processing Engine
PCM	Phase Changing Memory
IFM	Input Feature Map
OFM	Output Feature Map
DNN	Deep Neural Network
AIMC	Analog In-Memory Computing
MAMP	Mythic Analog Matrix Processors

The WiPLASH consortium is composed by

UPC	Coordinator	Spain
IBM	Beneficiary	Switzerland
UNIBO	Beneficiary	Italy
EPFL	Beneficiary	Switzerland
AMO	Beneficiary	Germany
UoS	Beneficiary	Germany
RWTH	Beneficiary	Germany

Contents

List of Figures

List of Tables

1. Introduction

Analog In-Memory Computing (AIMC) is an emerging paradigm holding promise to overcome the well-known von Neumann bottleneck by executing operations such as matrix-vector products in the analog domain within a crossbar arrangement, with millions of operations executed simultaneously. Both charge-based memory technologies (e.g. SRAM, DRAM, and flash), and resistance-based memory technologies (e.g. RRAM, PCM, and STT-MRAM) can serve as elements for such computational units [1].

Among several application domains, demonstrations of AIMC-based architectures have appeared in the field of Deep Neural Network (DNN) inference acceleration, showing outstanding peak energy efficiency in the order of hundreds of TOPS/W [1, 2]. An early market industrial example is represented by Mythic [3], claiming efficiency of 4 TOPS/W exploiting 8-bit flash-based Mythic Analog Matrix Processors (MAMP) arranged as a systolic array, all connected through a mesh topology network on chip. From a research perspective, several approaches claimed tens to hundreds of TOPS/W by exploiting several different approaches, with a quite diverse set of choices in levels of numerical precision and memory technologies [1, 2].

However, several fundamental challenges are still open to achieve the claimed levels: the intrinsic variability of analog computing both in the charge based and resistive domain [1]; difficulties in dealing with low-precision computations that are often the only ones supported by AIMC-based architectures [1]; the necessity of specialized training [4]; the poor flexibility of IMC, that is well matched only for a limited set of primitives such as matrix-vector multiplications [2]. As a result, most AIMC-based architectures fabricated so far have been demonstrated on trivial neural networks (up to ten layers) trained on single layers or simple data sets such as CIFAR-10 or MNIST [1], which are not representative of real-life, DNN-based applications.

This report focuses on the architectural challenges described above. To tackle the limited flexibility of AIMC-based computing, some architectures couple general-purpose processors to analog in-memory computing cores. This allows extending the functionality of In-Memory Accelerators (IMA) creating heterogeneous analog/digital computing tiles, connected to the system bus [3, 5]. However, performing linear operators with accelerators such as IMA moves the bottleneck of the computation to the digital part. For this reason, augmenting the heterogeneous cluster with a single core might not be sufficient to sustain the computing requirements of IMAs; moreover, low bandwidth and high communication latency between the processor and the IMA might form a remarkable bottleneck for heterogeneous computing.

This deliverable presents the design and modelling of an IMA, and the new paradigm for AIMC-based heterogeneous computing envisioned in WiPlash, where an IMA is integrated within a parallel tightly-coupled cluster of RISC-V processors. We present a design space exploration based on a key building block of the MobileNetV2 CNN, representative for a wide range of modern DNNs leveraging depthwise convolutions to reduce the size of the model by up to one order of magnitude with respect to first-generation models. We analyze the architectural bottlenecks for MobilenetV2 execution on such heterogeneous system. The IMA on itself can reach outstanding performance and efficiency peaks that are dictated by the size of the activated crossbar given the constant time to output for analog computations. But, the cost for auxiliary computation, data marshalling and inefficiency of depthwise layers from a significant constriction for efficiency (80%) suggesting to further extend these clusters with specialized digital accelerators better tuned for these functions that will be explored in the rest of WiPLASH project.

The rest of the report is organized as follows: In the first part of the report, the proposed system architecture containing the in-memory accelerator (IMA) is explained using a bottom-up approach. Chapter 2 will describe the operation of the In-memory computing PCM crossbar.The performance and energy modeling aspects at the crossbar level is discussed here. Then in Section 2.2 we dive into the details of PCM device level dynamics. The simulation models that were developed capturing these dynamics are discussed in Section 2.3. Chapter 3 briefly describes the heterogeneous cluster at the top level. Section 3.1 details the surrounding of the IMA crossbars and explains the key micro-architectural components: the controller, streamer and the engine. It also outlines interfacing architecture of the IMA subsystem with the rest of the cluster. The third part of the report, Chapter 4, is dedicated to a case study conducted using MobileNetV2 as an example application. Alternative programming strategies exploiting resources and performance tradeoffs in the IMA crossbar as well as those in rest of the cluster with varying degrees of workload distribution is studied to this end.

2. In-memory accelerator crossbar

In this section we introduce the crossbar architecture of the In-Memory Accelerator (IMA). It also covers how a basic matrix vector multiplication operation can be performed with the crossbar using in-memory computing techniques.

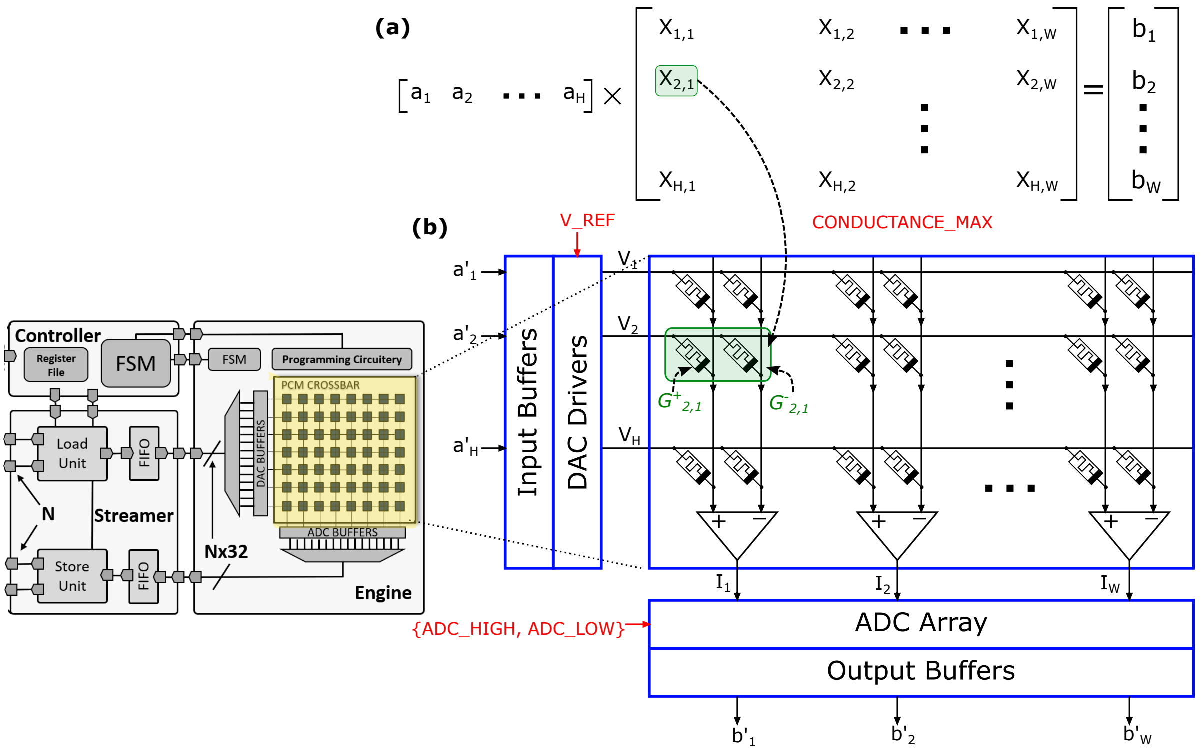


Figure 2.1: **(a)** standard matrix vector multiplication where vector **a** is multiplied with matrix **X** to output vector **b** (b) Illustration of matrix vector multiplication operation on differential PCM crossbar array. Its placement within the IMA subsystem is highlighted on the left.

As shown in Fig. 2.1, the crossbar supports programming elements of a matrix on the cross points of the crossbar, with a maximum matrix size of 1000 in each of the two dimensions due to fabrication limits of the crossbar array. Each cross point has a dual-Phase Changing Memory (PCM) device unit cell. A positive valued element in the matrix is programmed on the left PCM device of the unit cell with the corresponding conductance value while right PCM device is reset to a zero conductance value. A negative valued element in the matrix is programmed on the right PCM device of the unit cell while the left PCM device of the unit cell in this case is reset to a zero conductance value. The value of the matrix element is converted to PCM conductance value using the formula given in Eq. 2.1.

$$\{G^+_{i,j}, G^-_{i,j}\} = \begin{cases} \{round(\frac{7|X_{i,j}|}{X_{max}})\frac{G_{max}}{7}, 0\} & \text{if } X_{i,j} > 0 \\ \{0, round(\frac{7|X_{i,j}|}{X_{max}})\frac{G_{max}}{7}\} & \text{if } X_{i,j} < 0 \\ \{0, 0\} & \text{otherwise} \end{cases} \tag{2.1}$$

where $X_{i,j} \in \mathbb{R}$, $i \leq H$ height of the matrix, $j \leq W$ width of the matrix and $G^+_{i,j}, G^-_{i,j} \in \{0, \frac{G_{max}}{7}, \frac{2G_{max}}{7}, \frac{3G_{max}}{7}, \frac{4G_{max}}{7}, \frac{5G_{max}}{7}, \frac{6G_{max}}{7}, G_{max}\}$ takes one of the seven predefined multi-level conductance states or reset state. This allows packing 8 matrix element values into a 32-bit wide peripheral data bus in a single cycle - each value having 4 bits - a sign bit and 3bit magnitude.

The input vector **a** $\in \mathbb{R}^H$ before sending through the streaming interface is quantized to 8bit integer values as:

$$\begin{gathered} \textbf{a'} = round(\frac{\textbf{a}}{127}) * 127 \\ \text{where}\{\textbf{a'} \in \mathbb{Z}^H | -127 \leq \textbf{a'} \leq 127\} \end{gathered} \tag{2.2}$$

The quantized input values are converted to analog voltage levels using digital-to-analog converters (DAC) using $V_i = \frac{a'_i V_{REF}}{127}$, $0 \leq i \leq H-1$ where V_{REF} represents the maximum voltage a wordline is driven with. The Ohm's law is then applied to each resistive memory device located at cross points of the crossbar array. Transferring a current given by $i^{+/-}_{i,j} = V_i * G^{+/-}_{i,j}$ to the corresponding bitline. The currents from all devices connected to the same bitline add up according to the Kirchhoff's law to produce the total current on the j^{th} positive and negative bit lines which are then subtracted from each other to get the net total current as $I_j = \sum_{i=0}^{H-1} i^+_{i,j} - \sum_{i=0}^{H-1} i^-_{i,j}$.

Total net current is then passed to an analog to digital converter (ADC), which will quantize the current within the user-provided dynamic range $[ADC\_LOW, ADC\_HIGH]$ to an element in the output vector as given in

$$\begin{gathered} b'_j = round(\frac{127 * clip(I_j, ADC\_LOW, ADC\_HIGH)}{ADC\_HIGH - ADC\_LOW}) \\ \text{where } 0 \leq j \leq W-1 \text{ and } \{\textbf{b'} \in \mathbb{Z}^{\mathbb{W}} | -127 \leq \textbf{b'} \leq 127\} \end{gathered} \tag{2.3}$$

The output vector received from the crossbar **b'** is proportional to the original output of the matrix-vector multiplication (MVM) operation **b** and the proportionality constant is a function of maximum conductance G_{max}, reference wordline voltage V_{REF} and the ADC dynamic range $[ADC\_LOW, ADC\_HIGH]$.

2.1 Crossbar performance metrics

In Table 2.1 we present the performance metrics of the crossbar array.

Table 2.1: Performance metrics of the accelerator core

Name	**Value**
Total number of unit cells (2 device/unit cell)	HxW
Device programming levels	8
Bit equivalence of unit cell	1-bit sign, 3-bit magnitude
Core area	18.2xHxW μm^2
MVM core delay	70 ns
Output bandwidth	Wx8/70 $bits/s$
MVM energy per unit cell	50 fJ
MVM core energy	HxWx50 fJ
Core throughput	2xHxW/70 GOp/s
Core energy efficiency (unit cells only)	40 $TOP/s/W$
Core energy efficieny (including DACs/ADCs)	0.5x40 $TOP/s/W$

2.2 PCM device dynamics and modeling

[Responsible partner: IBM] In this section the dynamics of Phase-change memory (PCM) devices that are used to realize the in-memory compute array and the PCM model that is developed to fit these dynamics is discussed. PCM is arguably the most

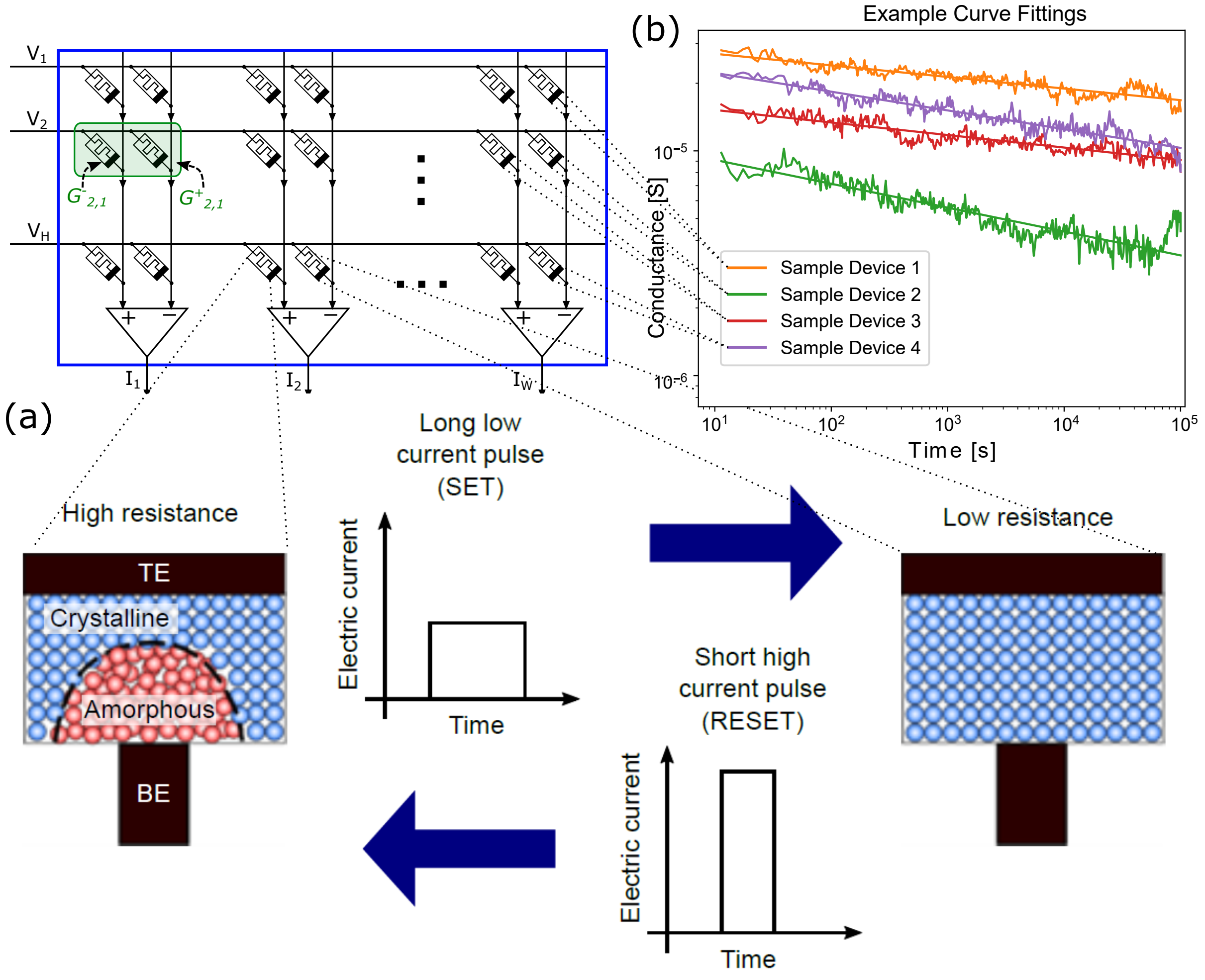


Figure 2.2: **(a)** Different 'phases' of PCM materials. On the right is the PCM device in low resistance crystalline phase, on the left it is in high resistance amorphous phase grown like a head of a mushroom. The phase can be shifted between either states by applying electrical pulses of specific shapes. **(b)** SET state measurements from 4 example devices, with their fitted curves

advanced memristive technology that has found applications in the space of computational memory [6], [7], [8]. A PCM device consists of a nanometric volume of a chalcogenide phase-change alloy sandwiched between two electrodes as shown in Fig. 2.2(a). The phase-change material is in the crystalline phase in an as-fabricated device see Fig. 2.2(a)(right). By applying a current pulse of sufficient amplitude (typically referred to as the RESET pulse) an amorphous region around the narrow bottom electrode is created via melt-quench process. The resulting "mushroom-type" phase configuration is schematically shown in Fig. 2.2(a)(left). The device will be in a high resistance state if the amorphous region blocks the conductance path between the two electrodes. This amorphous region can be partially crystallized by a SET pulse that heats the device (via Joule heating) to its crystallization temperature regime [9]. With the successive application of such SET pulses, there is a progressive increase in the device conductance. This analog storage capability and the accumulative behavior arising from the crystallization dynamics are central to the hybrid-precision in-memory MVM acceleration.

In order to model the most important PCM non-idealities, a simple conductance drift behavior has been assumed:

$$G(t) = G_{t_0} \cdot \left(\frac{t}{t_0}\right)^{-\nu} \tag{2.4}$$

where ν is the drift component and G means conductance after t time since programming. Since we fit these parameters to our measurements, we can simply chose reference time $t_0 = 1s$ so that Eq. 2.4 becomes

$$G(t) = k\, G_{max} \cdot t^{-\nu}. \tag{2.5}$$

We then introduce several parameters to model variations (see Table 2.2). The variations are assumed to be of Gaussian nature. Our final model of the conductance of a single PCM device is the following:

$$G(t) = \mathcal{N}(0, \tilde{G}_{\mathsf{r}}^2) + k\,(G_{max} \cdot \mathcal{N}(1, \tilde{G}_{\mathsf{p}}^2)) \cdot t^{-\nu \cdot \mathcal{N}(1, \tilde{\nu}^2)}, \tag{2.6}$$

with $k \in \{0, 1/7, 2/7..., 1\}$ is the fraction representing multi-level conductance, $\mathcal{N}(\mu, \sigma^2)$ being the normal distribution with mean μ and standard deviation σ and $\tilde{G}_{\mathsf{r}}^2$, $\tilde{G}_{\mathsf{p}}^2$, $\tilde{\nu}^2$ represent the variability in additive read noise, programming noise and drift respectively.

In order to obtain reasonable values for the above parameters, we program several devices on the PCM prototype chip [10] which includes 3 million PCM devices and fabricated with 90nm technology. The evolution of conductance values on these devices were then measures at time scales upto five orders of magnitude as shown in Fig. 2.2. A fit line is generated for each series of device measurements. The model parameters given in the table 2.2 are then estimated from the sample measurements.

Table 2.2: Estimated values of the model parameters.

Symbol	Description	Type	Value
G_{max}	mean SET conductance at time $t = 1s$	-	38.2×10^{-6} S
ν	mean drift exponent	-	0.0598
$\tilde{G}_{\mathsf{p}}$	programming variability	multiplicative	$31.7\,\%$
$\tilde{G}_{\mathsf{r}}$	read-out noise	additive	0.496×10^{-6} S
$\tilde{\nu}$	drift variability	multiplicative	$9.07\,\%$

2.3 Simulation models

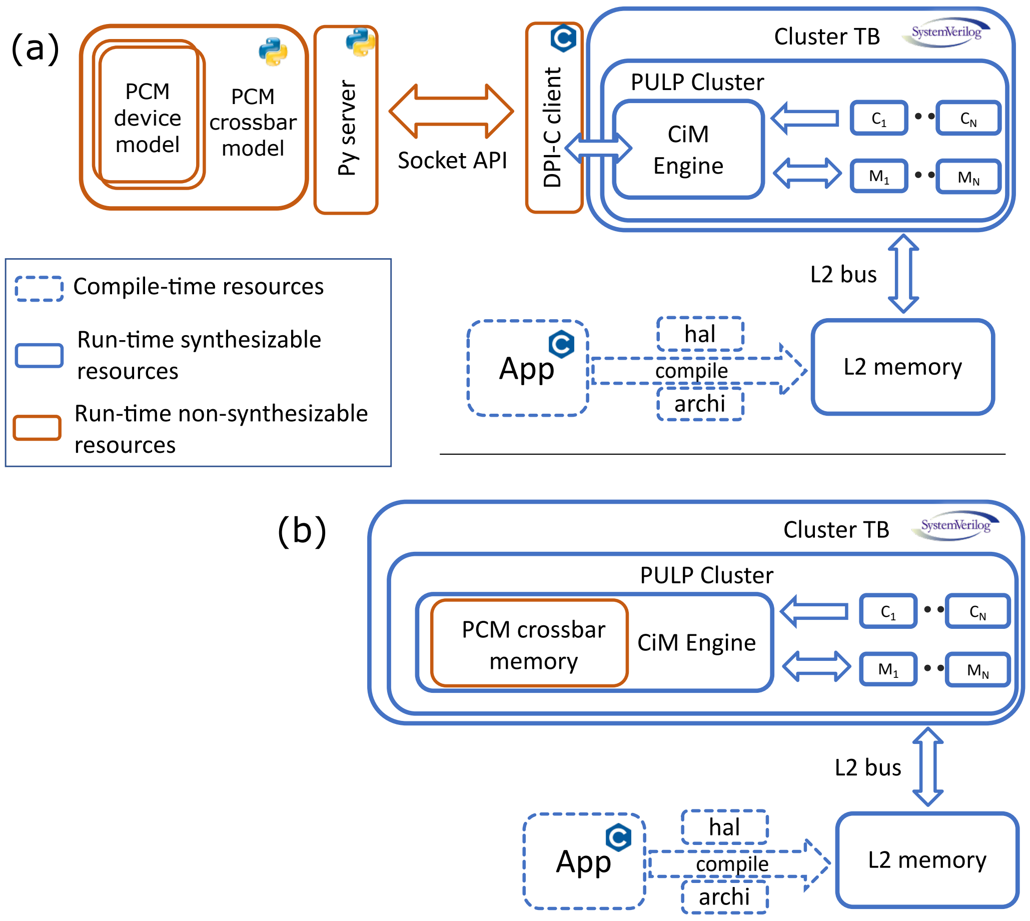


Figure 2.3: **(a)** Simulation framework in which PCM devices are more accurately modelled in python **(b)** Simulation framework in which PCM crossbar is modeled in SystemVerilog to achieve higher simulation speed at the expense of less accurate device conductance modeling.

We simulate the IMA integrated pulp cluster in two modes. In both modes, we use pulp tool chain and the runtime to build and run user application which is written in C.

1. In the first mode, the PCM model is in python. In these models we capture all the device dynamics discussed in section 2.2 using python computing packages such as numpy. We also include accurate programming and MVM operation delay information into these models. At the runtime a server is invoked on the python side and connected to a C-based client using BSD socket API [11]. The C-based client is then connected to the pulp cluster which is running on SystemVerilog using a direct programming C interface. We use a simple message passing protocol to send PCM read write and MVM evaluation requests from the SystemVerilog testbench side and exchange the timestamps to model the simulation delays.

2. In the second mode, the PCM crossbar is modeled entirely using a non-synthesizable SystemVerilog class. This reduces the simulation time considerably however at

the expense of less-precise modeling of PCM dynamics. The PCM crossbar array is modeled as an ideal storage unit. Nevertheless, the delays that are incurred during the access of the PCM crossbar memory are properly measured. This allows us to expand the realm of applications that we can simulation with IMA integrated pulp cluster to more practical workloads such as MobileNetV2. (see section 4.4. Further architectural optimizations and parameter tuning were conducted based on the findings of these simulations.

3. Heterogeneous IMA-based Computing Cluster

This chapter describes the integration of the IMA described in Chapter 2 within a tightly coupled cluster of RISC-V processors. The baseline cluster used within this project is based on PULP architecture [12] which structure can be seen in Fig. 3.1 on the left side of the figure. The cluster incorporates 8 RISC-V cores who share a single-cycle latency, word interleaved data memory called Tightly Coupled Data Memory (TCDM), or referred to as L1 memory. This memory configuration reduces the number of conflicts when multiple accesses are carried out by many sources at the same time (cores and accelerator). The core's ISA includes standard RISC-V RV32IMC [13] and a custom extension called Xpulp [14] that aims to accelerate arithmetic intensive kernels. This extension gives significant performance boosts when executing inner kernels of Convolutional Neural Network (CNN) where, hardware loops, packed SIMD dot products, post increment load & store, lands close to 9x performance w.r.t. RV32IMC (Fig 7 in [15]). The cores instructions are fed by a hierarchical instruction cache, where each core is supplied by a 512B private cache that is connected to a 4KB shared among all cores. In addition, most compute-heavy workloads can be offloaded to accelerators (In-Memory accelerator in this case) by accessing the internal control register file via peripheral interconnect and programming them based on the task structure.

3.1 IMA Subsystem Architecture

The IMA exposes a control and a data interface towards the rest of the cluster based on a standardized Hardware Processing Engine Hardware Processing Engine (HWPE) interface [1]. The data interface employs a direct connection with TCDM memory, composed of 16 parallel 32-bit banks in this work, through the same interconnect used by cores. The number of master ports is a design-time parameter N_{port} that can be chosen depending on the required bandwidth – as we show in Section 4.2, 4.4.

In Fig. 3.1, we show a detailed view of the IMA subsystem. The accelerator is composed of three main blocks. The *controller* includes the register file and the internal FSM coordinating the other blocks. The *engine* contains both the digital and analog parts of the IMA datapath. The digital part is composed of buffers for ADCs and DACs and control circuitry; the analog core encloses all the PCM devices (including PCM programming circuitry), as well as the ADCs and DACs themselves. The *streamer* block contains the address generators for memory transactions, implements the request protocols towards the TCDM, realigns data, and takes care of contentions. The address generators are capable of three-dimensional stridden access. This is extremely use-

[1] https://hwpe-doc.readthedocs.io/en/latest/

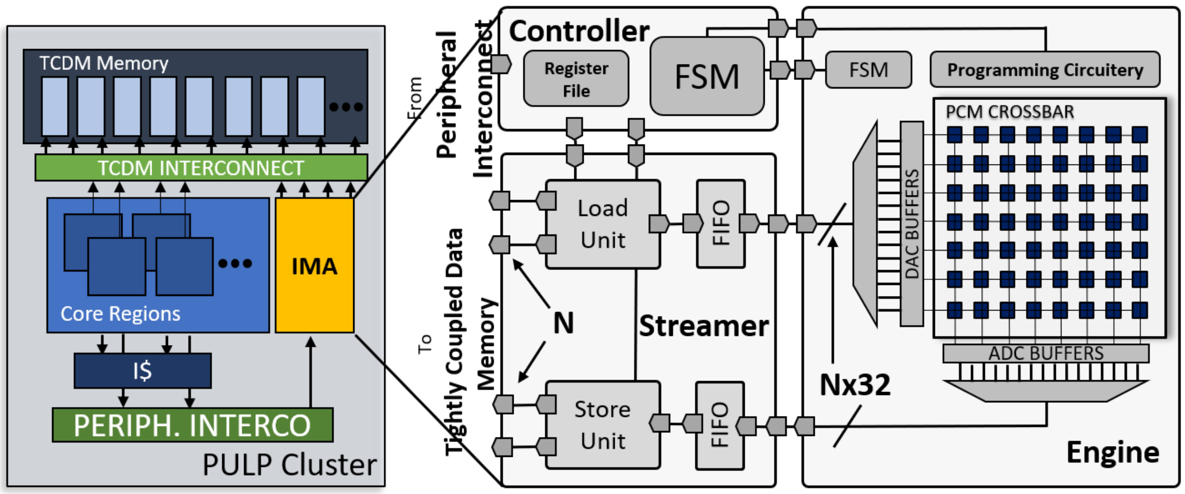


Figure 3.1: Heterogeneous Cluster and In-memory Accelerator subsystem.

ful in fetching the input tensors for CNNs where we have chunks of contiguous data separated between each other. These chunks have to be made contiguous inside the engine's data buffers and multiple stridden memory accesses allows this task to be completed without overheads. Data coming from N_{port} 32-bit TCDM ports are merged into a unique stream of data using a simple ready/valid handshake, which is fed to the engine. Conversely, data streams coming from the engine towards the TCDM memory are split in N_{port} 32-bit TCDM accesses.

The configuration sequence of the IMA starts when a core acquires a lock over the accelerator by reading a special ACQUIRE register through the peripheral control interface. After that, the core can interact with the IMA by: programming the PCM devices with the weights of one or multiple layers; reading the conductance value of a PCM device; programming a *job* by setting the address of input and output data in TCDM and the ADC configuration; when the configuration is over the job can be started by writing to a special TRIGGER register. To minimize IMA configuration and synchronization overhead, multiple jobs can be pipelined by setting the register file with the correct strides. Thus, a whole layer can be executed with only one configuration phase. The IMA works on input data stored in L1 with the HWC format, i.e., with consecutive data elements encoding pixels that are adjacents in the channel dimension. The execution of a job is divided into three phases: STREAMIN: fetch data from the TCDM that is then streamed to the engine's internal DACs buffers; COMPUTATION: analog computation on the crossbar and writing of the ADCs buffers; STREAMOUT: stream data from buffers back to the TCDM. In Fig. 3.2, we show how a CNN layer is mapped into the IMA and how the computational timeline is executed. For a standard convolutional layer, the STREAMIN phase also includes a virtual IM2COL transformation [15] (achieved with the multiple stridden address generator), which is performed directly by the streamers, enabling to remap all computation supported by the IMA to matrix-vector products of the form discussed in Section 2. As a consequence, the PCM array computes C_{out} Output Feature Map (OFM) from a complete input volume of $C_{in} \times K \times K$ pixels in a single operation, where $C_{in,out}$ indicate the number of channels and K is the filter size.

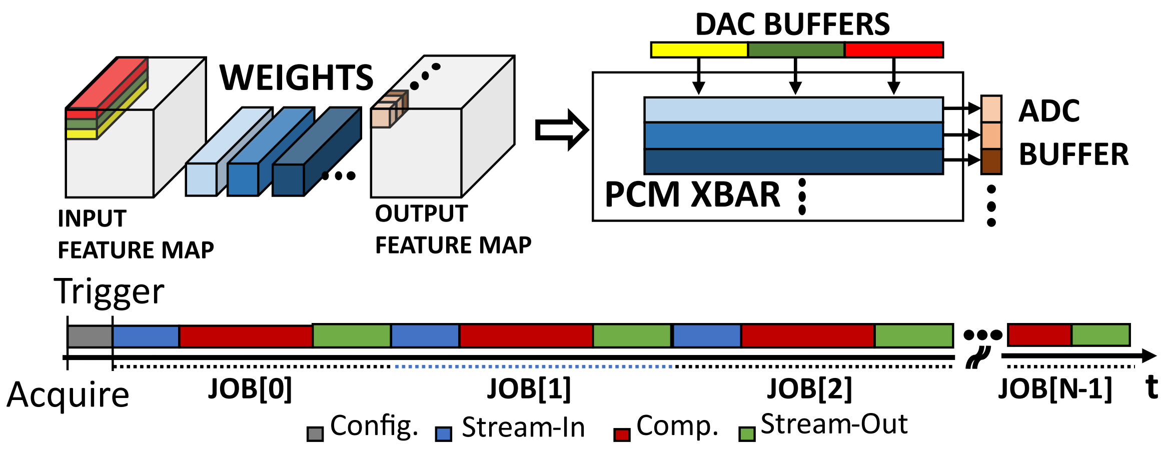


Figure 3.2: IMA mapping of standard convolutions on the PCM crossbar. Below a timeline of the execution model.

3.2 IMA Internal Registers

The internal registers includes all necessary configuration to allow a CNN layer to be exectuted in a single configuration phase. In Tab. 3.1 a list of all internal registers are presented with their function. In the context of the accelerator, a plot is the portion of the PCM Crossbar that you require to read or write to. In Fig. 3.3 on the left, we can see a visual representation of the different plot that can be programmed into the array, in this specific use case, each plot would be the parameters of the layer. To configure jobs and plots the state machine in Fig. 3.3 (right side) is inside the IMA controller which manage the data coming in through the peripheral interconnect.

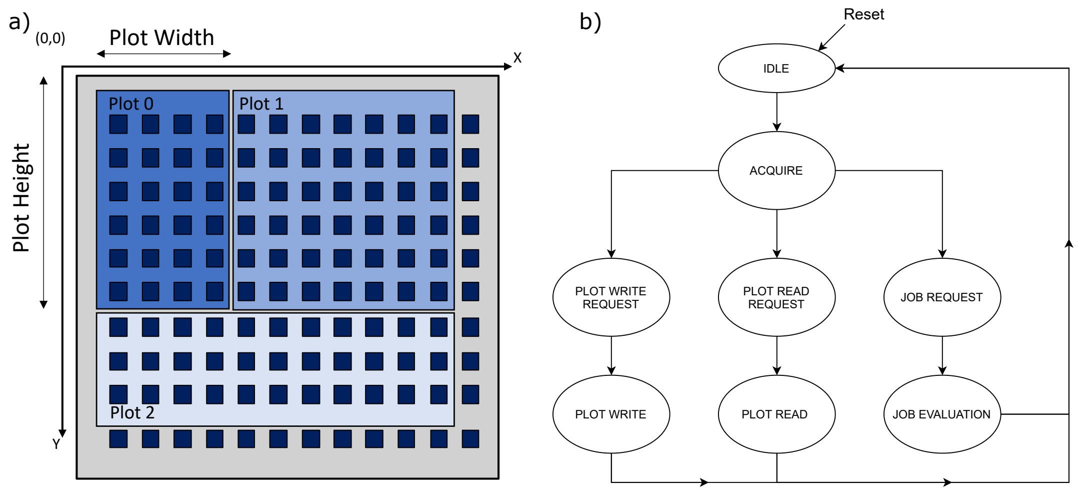


Figure 3.3: a): PCM Array subdivided in Plots, b): Controller FSM.

3.3 CNN configuration for IMA

In this section we will show how to configure the IMA accelerator to compute a CNN layer that has the same structure as the one that will be used for the performance eval-

Table 3.1: List of the programmable registers of the IMA.

Register Name	OFFSET	Descritpion
IMA_TRIGGER	0x00	Trigger the execution of a job (or jobs).
IMA_ACQUIRE	0x04	Acquire the accelerator control
IMA_FINISHED_JOBS	0x08	Return the number of concluded jobs since last read
IMA_STATUS	0x0c	Return the status of the IMA (1=busy, 0=idle)
IMA_RUNNING_TASK	0x10	Return ID of the currently running task. -1 if no task is running.
IMA_SOFT_CLEAR	0x14	Reset internal registers of the IMA (skipping the register file).
IMA_CHECK_STATE	0x18	Returns state of main FSM (transition diagram shown below). Useful for debugging purposes.
IMA_P_START_X	0x20	Coordinates of X where to start the plot.
IMA_P_START_Y	0x24	Coordinates of Y where to start the plot.
IMA_P_WIDTH	0x28	Plot width, used to count the number of expected parameters
IMA_P_HEIGHT	0x2c	Plot height, used to count the number of expected parameters
IMA_SUBMIT_PLOT	0x30	Request to write a Plot to the Crossbar array
IMA_J_START_X	0x34	X coordinate where the plot starts.
IMA_J_START_Y	0x38	Y coordinate where the plot starts.
IMA_J_WIDTH	0x3c	Width of the plot (output length)
IMA_J_HEIGHT	0x40	Height of the plot (input length)
IMA_J_SRC_ADDR	0x44	Address of the first input tensor element.
IMA_J_DST_ADDR	0x48	Address of the first output tensor element.
IMA_J_SRC_STRID	0x4c	Line Stride for the source tensor
IMA_J_DST_STRID	0x50	Line Stride for the ouptut tensor.
IMA_ADC_LOW	0x54	Low value of the ADC Dynamic Range
IMA_ADC_HIGH	0x58	High value of the ADC Dynamic Range
IMA_FETCH_LENGTH	0x5c	Number of memory transactions for the input tensor (depends on width of the input ports, e.g. 4 TCDM ports can fetch 4*4 bytes).
IMA_STORE_LENGTH	0x60	Number of memory transactions for the output tensor.
IMA_JOB_LINE_LENGTH	0x64	Number of bytes of the input line (Used for leftover).
IMA_JOB_FEAT_STRIDE	0x68	Second stride for the input tensor.
IMA_JOB_FEAT_LENGTH	0x6c	Number of times the first stride is used.
IMA_NUM_JOBS	0x70	Number of pipelined jobs.
IMA_ALPHA_IN_LENGTH	0x74	Number of jobs using the first job stride (input side).
IMA_ALPHA_IN_STRIDE	0x78	First stride for of the job pipelining (input side).
IMA_BETA_IN_LENGTH	0x7c	Number of jobs using the second job stride (input side).
IMA_BETA_IN_STRIDE	0x80	Second stride for the job pipelining (input side).
IMA_ALPHA_OUT_LENGTH	0x84	Number of jobs using the first job stride (output side).
IMA_ALPHA_OUT_STRIDE	0x88	First stride for of the job pipelining (output side).
IMA_BETA_OUT_LENGTH	0x8c	Number of jobs using the second job stride (output side).
IMA_BETA_OUT_STRIDE	0x90	Second stride for the job pipelining (output side).
IMA_JOB_LL_MEMT	0x94	Number of memory transaction for the input line (used for leftover).
IMA_JOB_FEAT_ROLL	0x98	Number of times the second stride is used.
IMA_DW_MODE	0x9c	Enable Depthwise Mode.
IMA_PR_ADDR_X	0xa0	Coordinates in X of the plot to read.
IMA_PR_ADDR_Y	0xa4	Coordinates in Y of the plot to read.
IMA_PR_VAL	0xa8	Return requested read data.

uation in Section 4.2. The configuration is mainly splitted in three phases: i) loading the parameters into the PCM Array; ii) Setup of individual jobs; iii) Setup of job pipelining. We note that the parameter phase would be done ”offline” and it is feasible during inference due to high delay for programming the PCM devices.
Submitting a plot: One of the core (core 0 for this example) will lock the accelerator by writing to the ACQUIRE register. At this step, plot submission starts when one of the plot register is written. Four registers are needed for setting the layer: IMA_P_START_X, IMA_P_START_Y will set the starting point for the plot (e.g.: in Fig. 3.3 Plot 0 would start at x=0 and y=0), while IMA_P_HEIGHT and IMA_P_WIDTH depends on the size of the filters (for this example $3 \times 3 \times 32$) and the number of output channel (64) respectively. Once these four register are set, the IMA expect a number of parameters that is equal to the $height \times width$ that has been just set. To write the parameters a 32-bit wide bus is used, so up to 8 weights can be written per request. The weights are written along the *X* axis first and then *Y*. For this example $(64 \div 8) \times (3 \times 3 \times 32)$ writes are required. Once the last parmater is sent to the crossbar the configuration phase ends with the IMA going back to IDLE state.
Setting up individual jobs: a job requires to use one of the plot that have been programmed in the previous phases. The registers IMA_J_WIDTH and IMA_J_HEIGHT are the same as the plot width and height so they must be consistent. It's also required to set the dynamic range of the ADC with the register IMA_ADC_LOW and IMA_ADC_HIGH (depends on the layer). Subsequently, we configure the IMA streamers so that they can extract an input tensor from the Input Feature Map (IFM) and streams it to the internal engine's buffer (same goes for the output streamer which takes the data from the ADC buffer and stream it back to memory). In Fig. 3.4 we can see how the input data is remapped into the IMA internal's buffers. The channels in the IFM are stored contiguously in memory (HWC format) as it can be seen by the relative indexes in the figure. To properly extract those input elements the steamers needs the strides to be set accordingly. For the input, we start by programming the line length and the number of memory transaction needed to fetch a complete line from memory (registers IMA_JOB_LINE_LENGTH and IMA_JOB_LL_MEMT respectively). From the point of view of the accelerator, a line is the amount of contiguous data in memory and for this examples is 32×3 (input channels times filter width). The IMA_JOB_LL_MEMT is used by the streamer to understand how many memory requests are needed to bring the input to the internal engine's buffer and it depends on the width of the stream (the stream width is a design time parameter that can be changed and represent how wide the interface to the memory is). In the example from Section 4.2, each line is $32 \times 3 = 96$ bytes long. If we were to assume 4 TCDM ports for the input interface (which are 4 bytes wide each), we would fetch 16 bytes per memory request meaning 6 will be programmed into IMA_JOB_LL_MEMT. The IMA_JOB_LINE_LENGTH register is used by the engine to deal with leftovers, this way, if our interface fetch more data that needed because the line is not a multiple of the width of the bus, the engine will discard the last bytes that are not used. Once the line length is set, its stride need to be configured by writing to IMA_J_SRC_STRID register. The line stride represent the distance between the lines. In the case we are looking at, the next line is $(16 - 3) \times 32$ bytes ahead, where 32 is the number channel in the IFM and 16 is the width of the image (we subtract 3 because of the size of the filter). Each input tensor is composed of 3 lines (or *FEAT* as referred in the name of the register) that is equal to the height of the filter (which will be programmed into

IMA_JOB_FEAT_LENGTH register). Last register to be configured for the input side is IMA_FETCH_LENGTH: this is the total number of memory transaction needed for the input tensor (still dependent on the number of input ports). It's value can be found by multiplying IMA_JOB_FEAT_LENGTH times IMA_JOB_LL_MEMT. We can now pass to the output streamer, which is easier to configure thanks to data being all contiguous in memory (see Fig. 3.4). This requires only to IMA_STORE_LENGTH register that is set to the number of memory transaction needed to store the data.

Setting multiple jobs: each job we execute will compute all the output channel

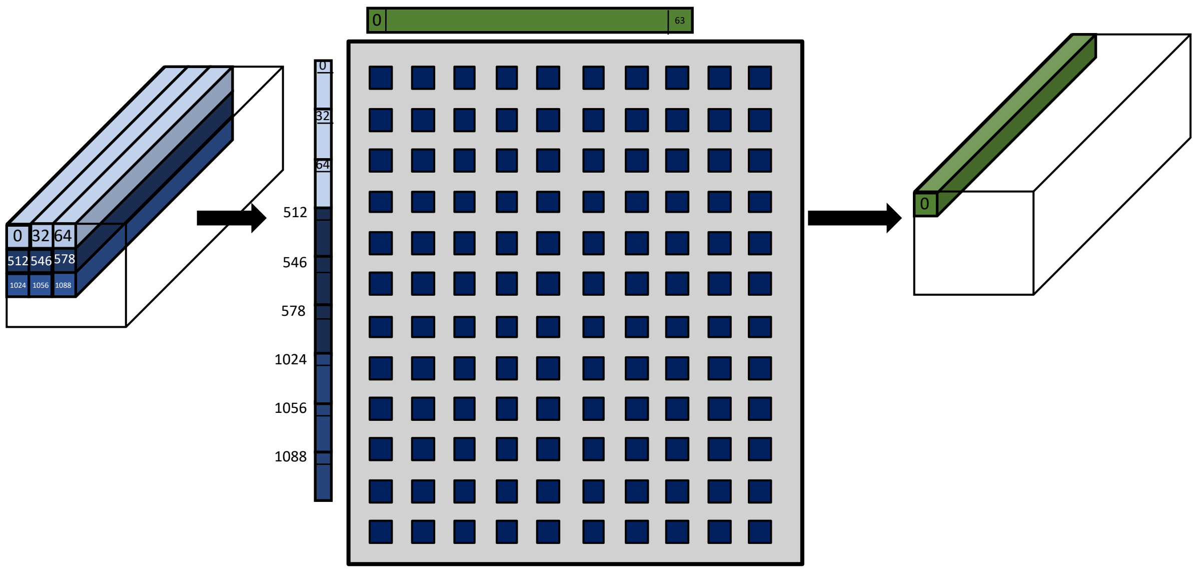


Figure 3.4: CNN remapping in the IMA.

for a given pixel (assuming that the width of the crossbar is wide enough to fit), meaning that we can compute the whole layer with a number of jobs that is equal to $Output\_layer\_width \times Output\_layer\_height$. This value will be set to IMA_NUM_JOBS register and is $16 \times 16 = 256$ in this example. Then, we need to set the strides of the input and output jobs by programming the *alpha* and *beta* registers. The alpha and beta register can be seen as two nested loop where the alpha is the inner one. On each job, alpha stride will be used until we reach alpha length, at this point, the beta stride is used and the beta counter is increased. This will go on until all jobs are completed. For the input, we have 16 jobs per row and a total of 16 rows meaning the value 16 we'll be written to both IMA_ALPHA_IN_LENGTH and IMA_BETA_IN_LENGTH. The alpha stride has to be set depending on the stride of the layer (the actual layer parameter) and the number of input channels. Given that the number of channels of the IFM is 32, the IMA_ALPHA_IN_STRIDE will be set to that (since the layer stride is 1). For the length we will use the width of the OFM width, which is 16. The beta length will be the same as input since width and height of the layer are the same. For the strides, we have to consider what happens at the end of each job. Here we have two cases, assuming we move from left to right: *we are inside the width of the IFM*, the next job moves to the right by 32 bytes (assuming stride 1) - alpha stride; *we reach the end of the width*, we need to move down along the height meaning that the next job is 32×3 bytes later (input channel times kernel width) - beta stride.

Once all these register are set we are ready to evaluate the layer, one of the core can acquire the accelerator and trigger it. The accelerator will compute the whole

OFM and go back to idle. In Fig. 3.5, a flow chart show the process of configuring and execution of the layer.

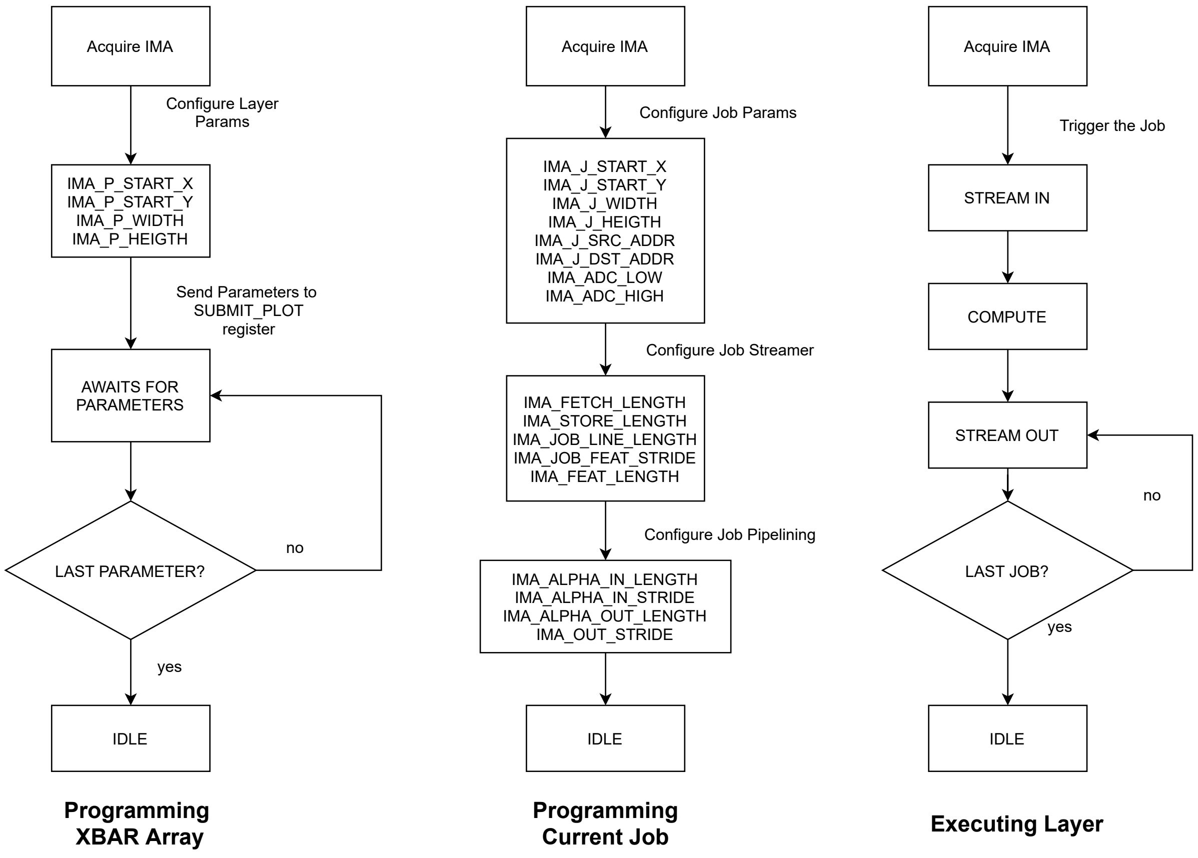


Figure 3.5: CNN flow-chart for programming, and executing a CNN layer.

4. Results & MobileNetV2 Case Study

4.1 Experimental Setup

The results shown in the next sections are attained by synthesizing the cluster described in previous section using Synopsys Design Compiler to target the GlobalFoundries 22nm FDX technology (SSG corner @ 0.59V and 250 MHz). For power analysis, we used Synopsys PrimeTime with typical corner with 0.65V at 25℃, with switching activity back-annotation from post-synthesis simulation. We assume that the PCM array is properly sized to fit all weights.

In the following Tab. 4.1 we report the implementation results showing the most notable blocks of the heterogeneous cluster. The first half of the table report the results in term of area, where we separated the analog (that is constant since we used a 256x256 PCM array for all the tests) and digital part. As expected, the increase of the memory interface bloats both the cluster interconnect and IMA subsystem while the rest remain constant. This effect can also be seen in the second half of the table where we report the power consumption while executing a CNN layer on the IMA. The power consumption does not only rise in the interconnect and IMA subsystem but also on the L1, where a bigger interface puts a heavier pressure on the memory subsystem. In Section 4.3 we'll see the effects of these results on the overall performance metrics we chose. From here on we assume the cluster frequency of 250MHz and using the convention 1 MAC = 2 OPs.

4.2 Baseline IMA performance

To assess the IMA's performance in a realistic baseline case, we used a standard convolutional layer with 3x3 filter, with 16x16 output size, 32 input channels, and 64 output channels (∼4.7 MMAC). The IMA went through several iterations where we optimized

Table 4.1: Implementation Results

AREA RESULTS [µm2]							
CONFIGURATIONS	**TOT AREA WITH IMA**	**TOT AREA**	**CORES AREA**	**IMA SUBSYSTEM**	**CLUSTER INTERCO**	**L1 TCDM**	**REST**
IMA - 1/1	1,612,701	422,701	11,439	29,530	10,481	118,971	252,280
IMA - 2/2	1,621,609	431,609	11,442	37,650	11,274	118,971	252,272
IMA - 4/4	1,637,745	447,745	11,439	52,217	12,837	118,971	252,281
IMA - 8/8	1,672,777	482,777	11,442	84,063	16,003	118,971	252,298
IMA 16/16	1,739,356	549,356	11,442	144,194	22,431	118,971	252,318
PCM 256x256	1.19 [mm2]						
POWER RESULTS [mW]							
Power Figures		**TOT POWER**	**CORE POWER**	**HWPE SUBSYSTEM**	**CLUSTER INTERCO**	**L1 BANKS**	**REST**
IMA - 1/1		3.84	0.35	0.66	0.08	0.74	2.00
IMA - 2/2		3.97	0.35	0.72	0.09	0.81	2.00
IMA - 4/4		4.07	0.35	0.76	0.10	0.85	2.01
IMA - 8/8		4.82	0.35	1.08	0.14	1.23	2.02
IMA 16/16		6.25	0.35	1.61	0.24	2.01	2.03

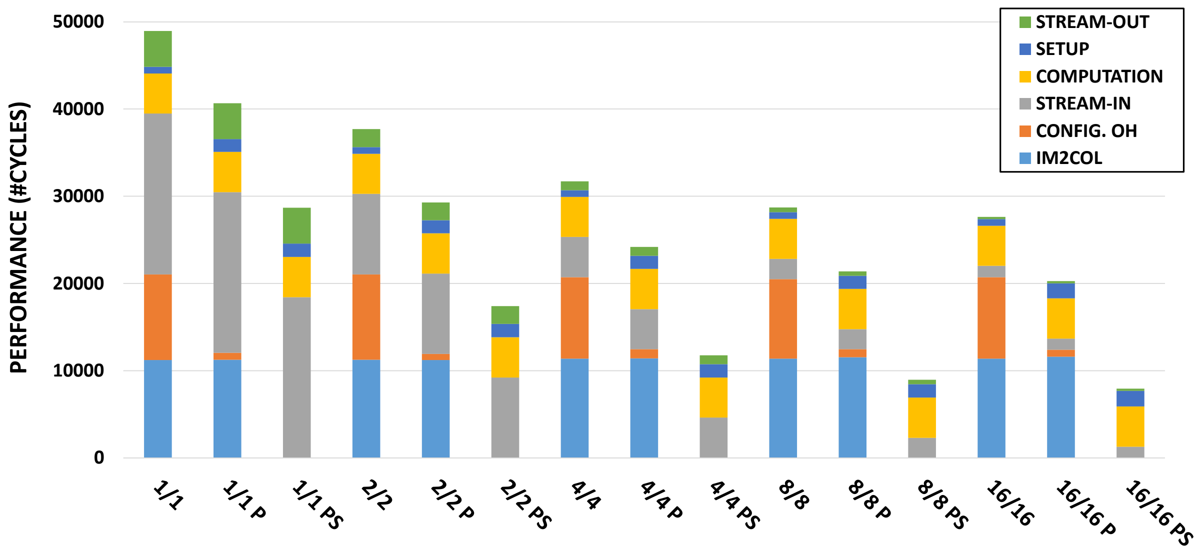


Figure 4.1: Performance between different IMA iterations on CNN layer.

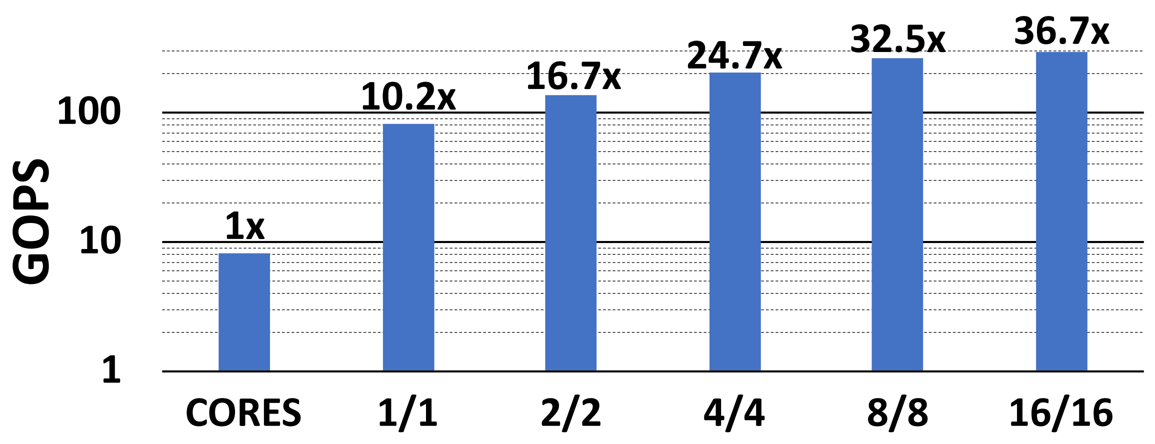
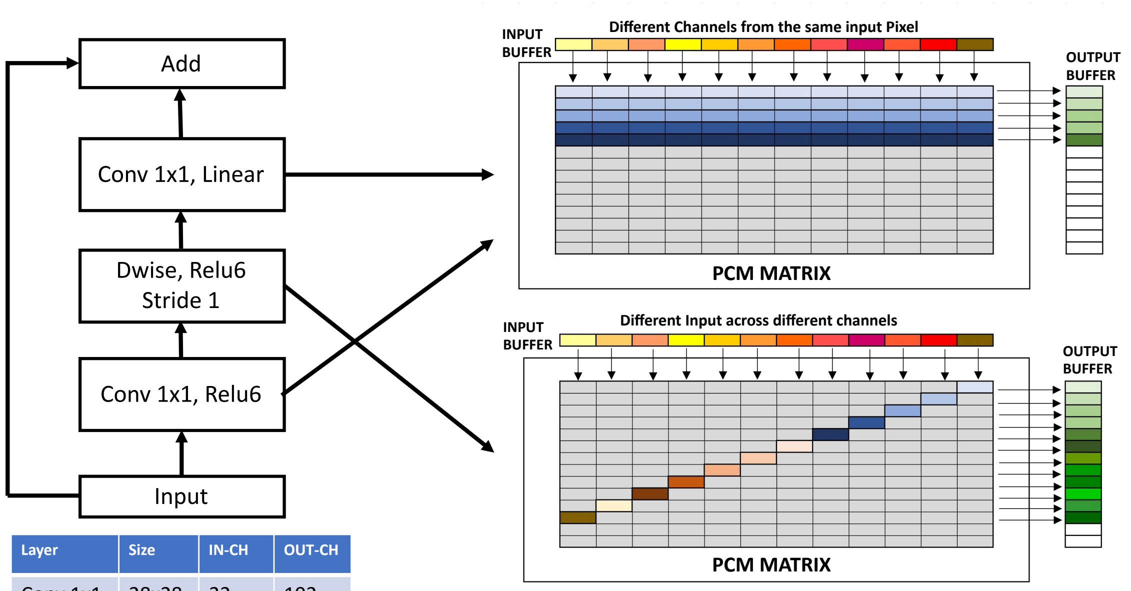


Figure 4.2: Performance on standard convolution: Software execution (SW) vs IMA acceleration (frequency = 250MHz). The N/N configuration indicates the number of load and store master ports.

configurations and data marshalling overheads. The first implementation used the 8 cluster cores as mean of data marshalling (creation of IM2COL [15]), this meant that the streamers would fetch data from a contiguous buffer and not leveraging the multiple stridden memory accesses. This implementation also suffered from heavy overheads due to configuration which was necessary between each job (no job pipelining). In figure 4.1 we can see the three implementation pitted against each other while varying the number of TCDM ports (e.g.: *1/1* means one port for input and one for output, each port is 32-bit wide) on the aforementioned layer. The job pipelining optimization is referred as *P* while *PS* drops the core for data marshalling and uses the multiple strides of the streamers.

Outside the Job's phases (stream-in, compute, stream-out) the IM2COL and the configuration overhead posed a significant burden on performance, which is the reason of their optimization. The SETUP, includes all overheads coming from FIFOs,



Layer	Size	IN-CH	OUT-CH
Conv 1x1	28x28	32	192
Dwise	28x28	192	192
Conv 1x1	28x28	192	32

Figure 4.3: Components of MobilenetV2 bottleneck block with stride = 1 and mapping structure in the PCM crossbar for depthwise and 1x1 layers. For depthwise, all the gray rectangles are padding required for computing more than 1 channel per job.

realignment of data, and address updates between jobs. In this particular case, the streaming of the input tensor was one of the main contributor that has been addressed by the increase in bandwidth. Fig. 4.2 shows the last IMA implementation compared with a pure software execution of the layer on the 8 cores using PULP-NN [15]. The speed-up ranges from 10.2x on 1/1 configuration up to 36.7x when 16/16 is used.

4.3 Case Study: MobilenetV2

To highlight the advantages and trade-offs of IMC on a realistic use case for extreme edge computing, we selected MobileNetV2, a widely used DNNDNN benchmark constructed as a deep stack of units called *BottleNecks*. For this analysis, we focus on the BottleNeck configuration that is shown in Fig. 4.3; the configuration is chosen so to fit the on-cluster TCDM (512 kB) without requiring any activation data tiling [16], where we note that this will further decrease energy efficiency and performance.

All layers in the BottleNeck can be mapped on the IMA. For the 1x1 convolutional layers, the mapping is direct as shown in Fig. 3.2 and Fig. 4.3, exploiting their high-level of channel parallelism. However, in the 3x3 depthwise layer each output channel depends only on a single input channel. This fact means that optimizing at the same time the array utilization and the execution performance is not possible.

A $K \times K$ depthwise layer with C in/out channels can be mapped as a standard layer with all weights out of a diagonal set to 0, as shown in Fig. 4.3. This means that out of $K^2 \times C^2$ crossbar locations, only $K^2 \times C$ are useful, leading to low utilization of the array. On the other hand, the depthwise can be split in separate jobs to reduce the array utilization, but this leads to a smaller amount of operations per job, reducing

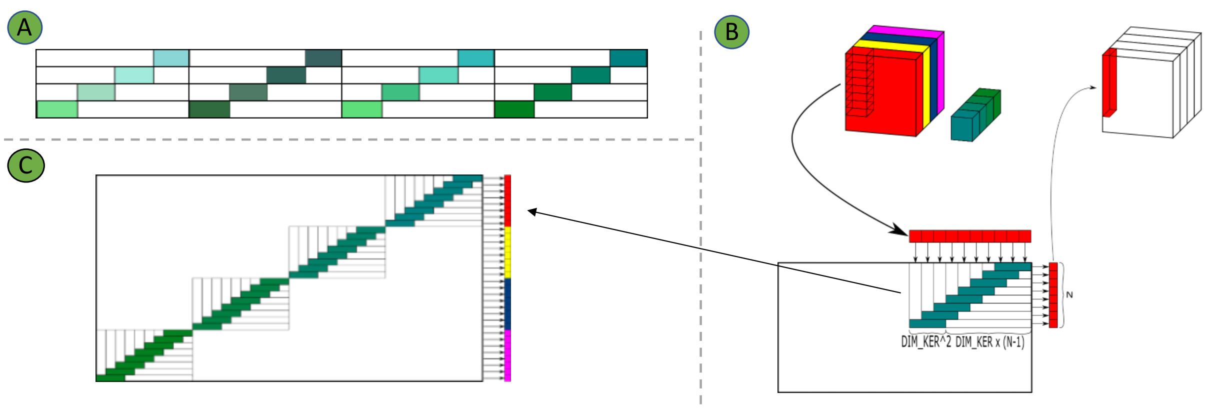


Figure 4.4: Show the different depthwise mapping: A) Shows 4 channel per job mapping; B) Shows multiple pixel per job mapping; C) Mix multiple channel per job with the mapping in B).

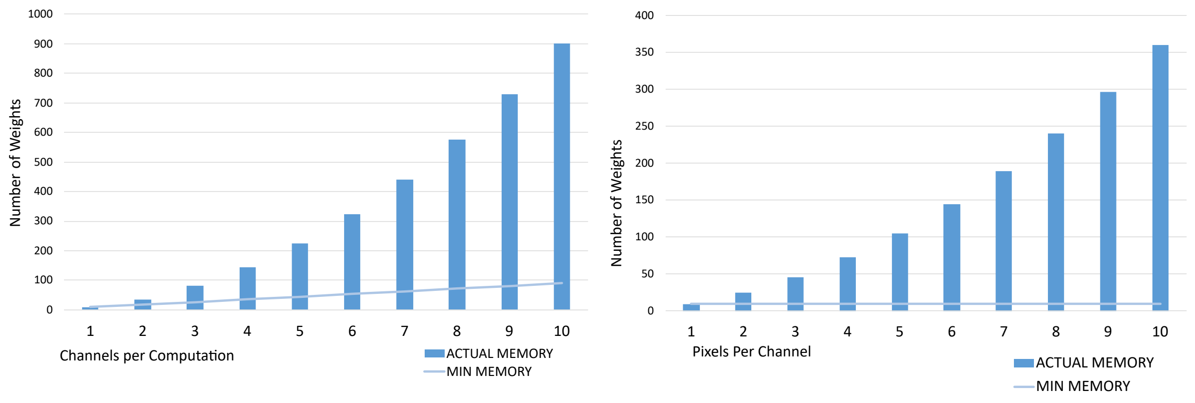


Figure 4.5: Mapping for depthwise on IMA. On the left: Number of Parameters vs Number of Channels per job. On the Right: Number of Parameters vs Number of pixels per job.

performance. The total number of crossbar elements required is in general given by $N_{xbar} = K^2 \times C \times C_{job}$, where C_{job} is the number of channels per job. For a MobileNet-V2, full throughput for all Bottlenecks would require a 23$\times$ larger array than what simply counting the number of parameters would suggest. This result stands even if the number of depthwise parameters is just $\sim$4% of the total number of weights.

Here, we considered $C_{job} = 8$ and 16 as reasonable trade-off configurations, which translates to an increase of 25% and 54% in the number of devices respectively. These are indicated as IMA8 and IMA16, respectively, in the following sections. We also explored a different mapping for depthwise layers as can be see in Fig. 4.4. In A) the mapping described above is shown, in this case there are 4 channel per job and different group of filters are put one next to each other. On B) we show a different way of mapping depthwise: we act on multiple pixels from the same channel instead of same pixel across channels. This representation could seem more densely packed than A) but we have to factor that it is computing only one channel per job. This means that we are storing the same filter of the same channel multiple times leading to a much lower PCM array utilization than A). In C) we show the results of mixing the solution B) with A) where we compute 4 different channels at the same time.

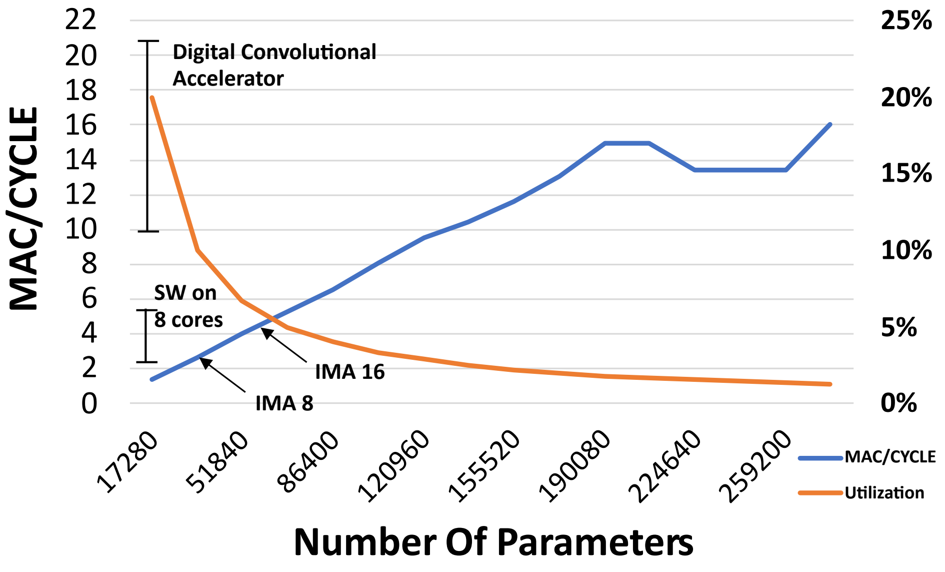


Figure 4.6: Thorough-put of the IMA on Depthwise layer vs. Amount of Parameters for a layer with 384 channels and 14x14 size. The orange line represent the utilization of the Array that is calculated by $number\_of\_weights \div total\_size$ were the total size includes the padding.

To better show the implication of these two approach the graphs from Fig. 4.5 represents the number of weights used while increasing the channels per job (if we refer to mapping A - left side) or pixels per job (referring to mapping B - right side). The utilization of the crossbar is low in both cases but is even worse in mapping B. This can be seen by looking at the *min memory* line, which represent the amount of memory needed to compute the pixels/channels (i.e.: for multiple pixels of the same channel one filter is needed, if we take 3×3 kernel as example this would mean 9 weights. In case of multiple channels the amount of min memory necessary would be $3 \times 3 \times num\_channel$). The rest of the work will consider the mapping of depthwise on the IMA as mapping A). On this we made a model that estimates the performance that could be expected on the IMA varying the number of channels per job and is shown in Fig. 4.6 (on the X axis the number of parameters is shown, which is directly related to channels per job) where we considered a TCDM interface 64 bytes wide. We can see performance scales quite linearly while the percentage of utilization drops significantly from the beginning (where it starts with 5 channels per job). The drops in performance going to the right is caused by the increase of data for the stream-in/out phase. If the size of the TCDM interface was smaller, the ”dent” would move to the left so this represent a best case scenario when we max out the memory bandwidth to the accelerator. Given the massive increase in number of parameter for IMA 16 and IMA 8 going up the curve would only exacerbate this effect.

An alternative solution supported by the heterogeneous cluster we propose is the parallel execution of the depthwise layer via software [15] on the 8 RISC-V cores of the cluster, intermixed with IMA-based execution of 1x1 layers. This configuration, which is reported as HYBRID, requires the parameters from the depthwise layer to be stored in memory instead of IMA which we consider a reasonable trade-off since those parameters account only for 4% of the total weights.

4.4 MobileNetV2 Bottlenecks Results

The performance results in this section are from the Bottleneck with sizes reported in Fig. 4.3 sweeping across 1 to 16 ports for stream-in and out. In Fig. 4.7 we can see how the benefits of adding TCDM master ports start to fall off after 4/4: the depthwise layer dominates the number of cycles (see Fig. 4.10) and increasing ports doesn't render as sizeable an effect as shown in Fig. 4.2. In particular, for the HYBRID solution, increasing bandwidth toward IMA with more ports does not influence the depthwise execution. In IMA16 configuration the bandwidth for depthwise saturates when all the channels can be fetched in one cycle: 4 TCDM ports of 4 bytes each are enough; going over only benefits 1x1 convolutions. The same reasoning can be applied to 8 channels per job, where 2 ports are sufficient.

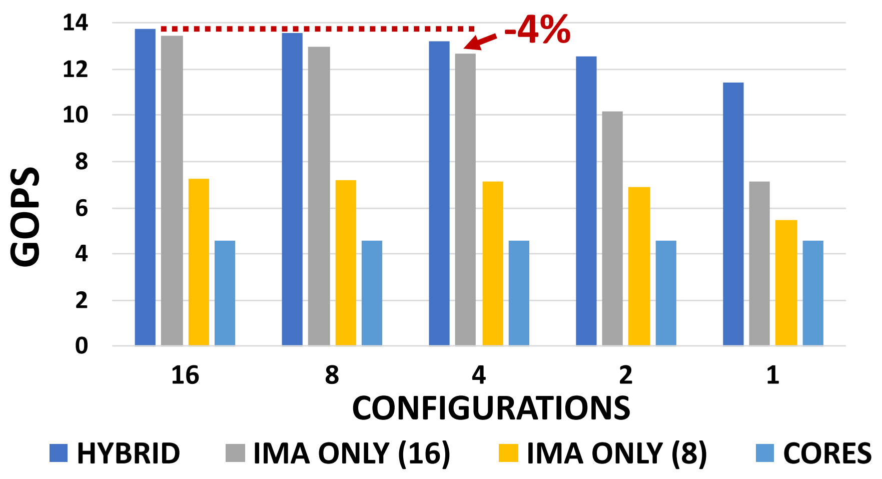


Figure 4.7: Performance results measured in GOPS. The arrow indicates the performance difference with the most efficient implementation.

Thus, the importance of the depthwise layer in the Bottleneck drives the total improvement when using the IMA down to ~3x the software implementation (down from ~36x on standard convolutions). Overall, the HYBRID configuration stands out as the fastest: this is because even in the IMA16 configuration, the depthwise layer is slower than in software, as can be seen in Fig. 4.10. Similar considerations can be made with respect to energy efficiency, noticing that adding more ports than necessary reduces energy efficiency (Fig. 4.8) with respect to the peak at 4 (HYBRID/IMA16) or 2 ports (IMA8), as it puts more pressure on the memory system.

To put in perspective the cost of increasing the throughput using IMA, the area efficiency reported in Fig 4.9 is relative to the effective area of the PCM arrays utilized to implement the Bottleneck (including padding). The HYBRID solution has the best result requiring ~3.25x and 2.13x smaller PCM area for the same bottleneck when compared to IMA16 and IMA8, respectively. Considering also the area of the cluster itself, we obtain 1.82x and 2.56x better GOPS/mm^2, respectively.

Last graph in fig 4.10 we show the average number of cycles per operation it takes to complete the bottleneck with 4 TCDM port configurations at 250 MHz. The components are split into sections as shown in Fig. 4.3. Dominating the computation time when using the IMA is depthwise, increasing the number of output channels reduces

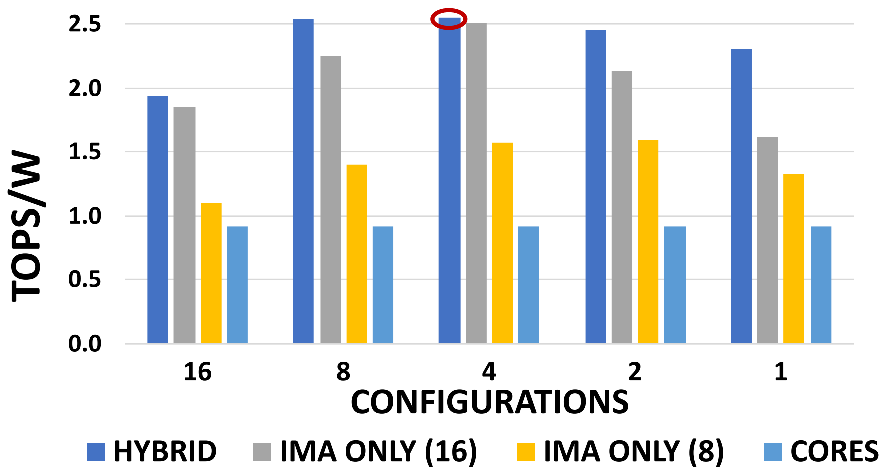


Figure 4.8: Performance results measured in TOPS. The circle highlights the top performing configuration.

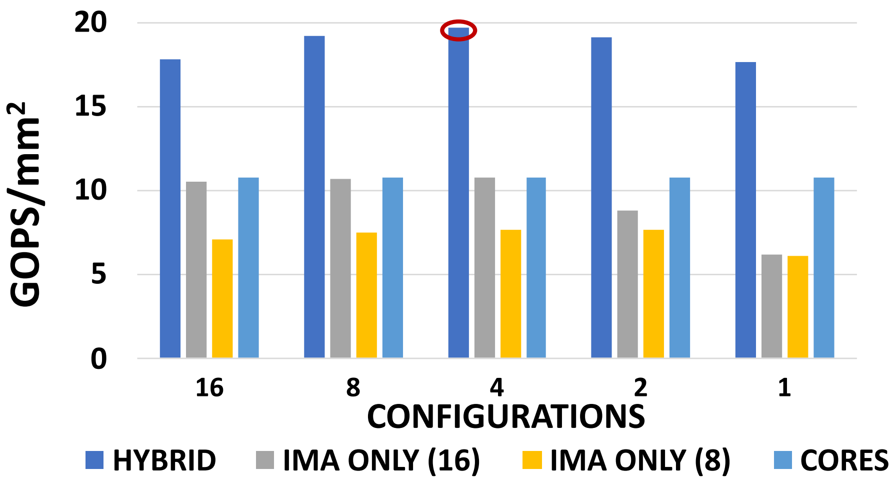


Figure 4.9: Performance results measured in GOPS/mm^2. The circle highlights the top performing configuration.

the number of jobs required, and it scales quite linearly given the fact that the output of the PCM crossbar requires a fixed time to evaluate the output. When looking at the convolutions 1x1, the amount of speed-up achieved is comparable with the Fig 4.2 when using the software version as reference. This graph shows that significant performance improvements can still be achieved if specialized digital accelerators tuned to perform key kernels such as depthwise layers are instantiated in the system.

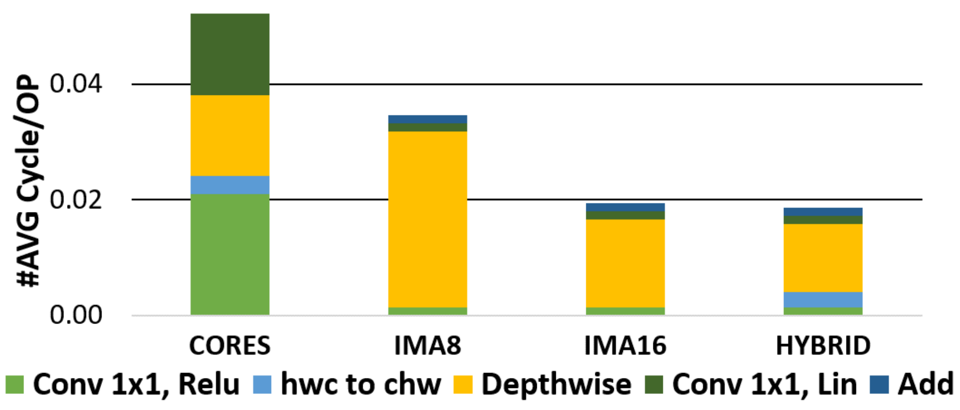


Figure 4.10: Impact on performance of the various Bottleneck phases (hwc to chw only needed on SW variants for depthwise). Results are taken using 4/4 port configurations at 250 MHz.

5. Conclusion

In this deliverable we described an In-Memory Accelerator (IMA) and its integration into a cluster of 8 RISC-V cores. As expected, the IMA boosts performance in matrix-vector multiplication based kernels such as convolutional layers of neural networks by a significant factor (up to 36x when compared to an 8-cores cluster in our experiments). We also show that the inflexible Matrix-Vector product paradigm imposed by IMAs requires some mitigation on the architectural side. This observation strongly motivates our choice to couple a highly efficient IMA with a highly flexible cluster of cores. In fact, even a relatively simple Bottleneck layer from a MobileNetV2 includes blocks that are not well-mapped to the IMA, specifically, depthwise separable convolutions. We show several possible mappings trading off area and performance, demonstrating that executing depthwise layers directly in the cores yields up to 2.56x better area efficiency without overhead in performance and energy. The heterogeneous system achieves 13.2 GOPS, 19.7 GOPS/mm^2 and 2.55 TOPS/W on a 4/4 configuration that is competitive with declared metrics from state-of-the-art academic [5] and commercial systems [3]. We argue that enhanced architectural heterogeneity is the key to fully exploit the potential of IMC architectures by offsetting their current limitations. Our future work includes further extending heterogeneous clusters with digital accelerators tuned to key kernels that are not well suited to IMC, such as depthwise layers, nearing the 100 TOPS/W targets in real-world DNN inference, as well as exploring system-level aspects such as cluster to cluster communication exploiting THz wireless transceivers developed in WiPlash.

Bibliography

[1] A. Sebastian *et al.*, "Memory devices and applications for in-memory computing," *Nature nanotechnology*, vol. 15, no. 7, pp. 529–544, 2020.

[2] N. Verma *et al.*, "In-memory computing: Advances and prospects," *IEEE Solid-State Circuits Magazine*, vol. 11, no. 3, pp. 43–55, 2019.

[3] D. Fick and M. Henry, "Analog computation in flash memory for datacenter-scale ai inference in a small chip," in *Hot Chips*, 2018.

[4] S. R. Nandakumar *et al.*, "Mixed-precision deep learning based on computational memory," *Frontiers in Neuroscience*, vol. 14, p. 406, 2020.

[5] H. Jia *et al.*, "A programmable heterogeneous microprocessor based on bit-scalable in-memory computing," *IEEE Journal of Solid-State Circuits*, vol. 55, no. 9, pp. 2609–2621, 2020.

[6] M. Cassinerio, N. Ciocchini, and D. Ielmini, "Logic computation in phase change materials by threshold and memory switching," *Advanced Materials*, vol. 25, no. 41, pp. 5975–5980, 2013.

[7] A. Sebastian, T. Tuma, N. Papandreou, M. Le Gallo, L. Kull, T. Parnell, and E. Eleftheriou, "Temporal correlation detection using computational phase-change memory," *Nature Communications*, vol. 8, no. 1, pp. 1–10, 2017.

[8] M. Le Gallo, A. Sebastian, G. Cherubini, H. Giefers, and E. Eleftheriou, "Compressed sensing with approximate message passing using in-memory computing," *IEEE Transactions on Electron Devices*, vol. 65, no. 10, pp. 4304–4312, 2018.

[9] A. Sebastian, M. Le Gallo, and D. Krebs, "Crystal growth within a phase change memory cell," *Nature communications*, vol. 5, no. 1, pp. 1–9, 2014.

[10] G. Close, U. Frey, M. Breitwisch, H. Lung, C. Lam, C. Hagleitner, and E. Eleftheriou, "Device, circuit and system-level analysis of noise in multi-bit phase-change memory," in *2010 International Electron Devices Meeting*, pp. 29–5, IEEE, 2010.

[11] J. Frost, *Bsd sockets: A quick and dirty primer*. Jim Frost., 1990.

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

[13] A. Waterman *et al.*, "The risc-v instruction set manual. volume 1: User-level isa, version 2.0," tech. rep., California Univ Berkeley Dept of Electrical Engineering and Computer Sciences, 2014.

[14] M. Gautschi *et al.*, "Near-threshold RISC-V core with DSP extensions for scalable IoT endpoint devices," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 10, pp. 2700–2713, 2017.

[15] A. Garofalo *et al.*, "Pulp-NN: accelerating quantized neural networks on parallel ultra-low-power risc-v processors," *Philosophical Transactions of the Royal Society A*, vol. 378, no. 2164, p. 20190155, 2020.

[16] A. Burrello *et al.*, "DORY: Automatic End-to-End Deployment of Real-World DNNs on Low-Cost IoT MCUs," *arXiv preprint arXiv:2008.07127*, 2020.