Project:

MNEMOSENE

(Grant Agreement number 780215)

"Computation-in-memory architecture based on resistive devices"

Funding Scheme: Research and Innovation Action

Call: ICT-31-2017 "Development of new approaches to scale functional performance of information processing and storage substantially beyond the state-of-the-art technologies with a focus on ultra-low power and high performance"

Date of the latest version of ANNEX I: 17/09/2020

D4.7– Refined CIM microarchitecture

Project Coordinator (PC):	Prof. Said Hamdioui Technische Universiteit Delft - Department of Quantum and Computer Engineering (TUD) Tel.: (+31) 15 27 83643 Email: S.Hamdioui@tudelft.nl
Project website address:	www.mnemosene.eu
Lead Partner for Deliverable:	IMEC
Report Issue Date:	31/10/2020

Document History
(Revisions – Amendments)

Version and date	Changes
07-05-2020	First draft version
30-06-2020	Second draft version
26-11-2020	Third and full draft version
22-12-2020	Final version reviewed by project coordinator

Dissemination Level

PU	Public	**X**
PP	Restricted to other program participants (including the EC Services)	
RE	Restricted to a group specified by the consortium (including the EC Services)	
CO	Confidential, only for members of the consortium (including the EC)	

European Commission

The MNEMOSENE project has received funding from the European Union's Horizon 2020 Research and Innovation Programme under grant

The MNEMOSENE project aims at demonstrating a new computation-in-memory (CIM) based on resistive devices together with its required programming flow and interface. To develop the new architecture, the following scientific and technical objectives will be targeted:

- Objective 1: Develop new algorithmic solutions for targeted applications for CIM architecture.
- Objective 2: Develop and design new mapping methods integrated in a framework for efficient compilation of the new algorithms into CIM macro-level operations; each of these is mapped to a group of CIM tiles.
- Objective 3: Develop a macro-architecture based on the integration of group of CIM tiles, including the overall scheduling of the macro-level operation, data accesses, inter-tile communication, the partitioning of the crossbar, etc.
- Objective 4: Develop and demonstrate the micro-architecture level of CIM tiles and their models, including primitive logic and arithmetic operators, the mapping of such operators on the crossbar, different circuit choices and the associated design trade-offs, etc.
- Objective 5: Design a simulator (based on calibrated models of memristor devices & building blocks) and FPGA emulator for the new architecture (CIM device combined with conventional CPU) in order demonstrate its superiority. Demonstrate the concept of CIM by performing measurements on fabricated crossbar mounted on a PCB board.

A demonstrator will be produced and tested to show that the storage and processing can be integrated in the same physical location to improve energy efficiency and also to show that the proposed accelerator is able to achieve the following measurable targets (as compared with a general purpose multi-core platform) for the considered applications:

- Improve the energy-delay product by factor of 100X to 1000X
- Improve the computational efficiency (#operations / total-energy) by factor of 10X to 100X
- Improve the performance density (# operations per area) by factor of 10X to 100X

Table of Contents

Preface

In WP 1, promising applications for a CIM architecture were investigated. These applications are widely used in many platforms and require processing large amounts of data. The potential improvement in terms of energy and performance were highlighted in the same work package. WP2 and WP3 focused on the (automatic) extraction of kernels suitable for execution on a CIM accelerator and the definition of a system-on-chip encompassing modern-day state-of-the-art compute units and integrating them with the CIM accelerator, respectively. While WP4 mainly focused on the memristor technologies to implement application relevant operations (identified in WP1), there was still a gap between the works in WP2, WP3, and WP4. In Deliverable D4.6, we proposed a simulator to fill this gap and we presented the initial CIM tile architecture, the CIM simulator, as well as an initial investigation into potential performance and energy results of this initial CIM tile architecture. In this deliverable (D4.7), we further refined the CIM tile architecture, adapted the CIM simulator accordingly and present more detailed performance and energy results and demonstrate how the design-space exploration (between different memristor technologies) can be performed using our CIM simulator.

1. Introduction

Emerging applications such as neural networks, databases, and image processing are widely used in different platforms including real-time embedded systems, back-end data centers, etc. These applications require processing large amounts of data that is usually located far away from the processing units. Consequently, performance and power consumption of the systems which fulfill these processing needs have become crucial and draw increasingly more attention. Currently, traditional von-Neumann architectures have been stretched to attain adequate performance at reasonable energy levels but are clearly showing limitations for further improvements. The main limitation is the conceptual separation of the processing unit and its memory, which makes the data movement between memory and processing unit the main performance and energy bottleneck. The solution to overcome this is to employ the Computation-in-Memory (CIM) approach that proposes the utilization of (new) technologies that allows for both storage and computing within the same (storage) structure. This is achieved by exploiting special characteristics of emerging non-volatile memories called memristors such as resistive RAM (ReRAM), phase change memory (PCM), and spin-transfer torque magnetic RAM (STT-RAM). No matter which technology is used for fabrication, memristor technology has great scalability, high density, near-zero standby power, and non-volatility. Accordingly, memristor technology with the aforementioned characteristics opens up new horizons toward new ways of computing and computer architectures.

Until now, the main focus of researchers was to enhance the characteristics of memristor devices such as latency and endurance [1] [2] [3] [4] [5]. Researchers have already proposed different innovative circuit designs based on memristor devices to exploit their capabilities of co-locating computation and storage together [6] [7] [8] [9] [10] [11] [12] [13]. Moreover, within a single memory array as well as at inter-array level huge parallelism can be flexibly achieved as each memory tile becomes a powerful computation core. It was demonstrated that due to these two main features of memristor-based designs, significant energy and performance improvement can be gained [14]. It is widely accepted that the dot-product (and, in turn, the matrix-matrix multiplication) operation is the most suited for the memristor-based designs. Consequently, convolutional and deep neural network are the potential applications that have been widely studied by the researchers to exploit memristive crossbar structures [15] [16] [17] [18] [19] [20]. However, researchers have also proposed other types of operations, e.g., Boolean operations [21] [22] [23] or arithmetic operations like additions [24]. More information

on available research regarding device characteristics and potential applications for in-memory computing can be found in [25] [26].

Having said this, **there is no work in the research community that allows for easy comparison of these supported operations at the application-kernel level between different technologies nor is it possible to emulate complex operations**, e.g., matrix-matrix multiply, when the underlying technology does not allow for direct implementation. Furthermore, the **interactions between the analog memory array and its supporting digital periphery is largely overlooked**. Efficient organization of peripheries is crucial. Otherwise, it alleviates the energy gain achieved by the memory crossbar. As an example, computation over integer numbers is required by most applications. However, since limited levels can be stored in one memristor cell, a number should be distributed over cells, which requires some extra processing outside the crossbar to get a meaningful result. This clearly shows the importance of organization for periphery circuits. In our prior work, presented in Deliverable 4.6, a new instruction set architecture (ISA) is introduced with the objectives of (1) orchestrating digital and analog components of memristor tiles and (2) bridging the gap between high-level programming languages and the CIM architecture. Our compiler written in C++ generates low-level instructions based on high-level kernels, which are supposed to execute on the CIM tile. The compiler is aware of the architecture configuration, the technology constraints, and the datatype size requested by the application. In addition, in order to accomplish design space exploration targeting performance and energy, we designed a modular simulator written in SystemC.

In this report, we describe:

1. an extension of our CIM-tile architecture to allow operations in the digital periphery and the analog array operate in parallel through pipelining. The pipelined stages are unbalanced due to the analog components which can have widely varying latencies.
2. our proposed addition unit, explained in D4.5, and evaluate it using our in-memory simulator. This addition structure was tailored for crossbar array to aid an in-memory crossbar to perform additions targeting integer matrix-matrix multiplication (MMM). The proposed design utilizes minimum-sized adders and is customizable in order to support varying numbers of ADCs.
3. how our in-memory simulator produces energy numbers for our tile components including crossbar. In order to have a more accurate number for the crossbar, the simulator enhanced to produce a data-dependent energy number.
4. the special features and optimizations of the STT-MRAM crossbar for binary logic and MMM operations.

2. Background on Initial CIM-tile architecture

In this section, we provide a short summary of what was presented in D4.6. We briefly explain the CIM-tile architecture and its analog as well as digital components. In addition, the nano-instructions defined to organize the interactions between these components will be reviewed.

2.1. Overall Tile architecture

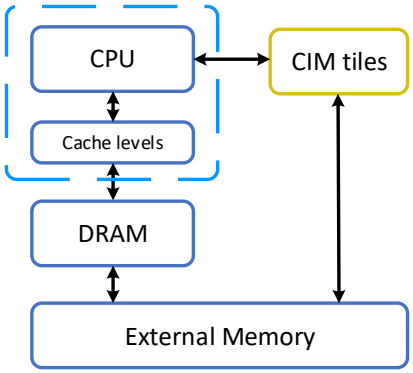

Figure 1: Potential high-level computer architecture using CIM tiles

A CIM tile can be either employed as a standalone accelerator or integrated to the conventional computer architecture. In the latter case, there are different ways to employ CIM tiles. Figure 1 depicts one potential way in which a CIM tile can be seen as an off-/on-chip accelerator from the CPU. As mentioned earlier, we focus on the CIM-P 1T1R structure in which the (computational) results of the (memory) array operations are captured in the (digital) periphery. Figure 2 presents the architecture of the CIM tile that includes the required components and signals which can control digital or analog data. The operations that can be executed on the crossbar are divided into two categories: 1) write and, 2) read and computational operations. The computational operations include addition, multiplication, and logical operations.

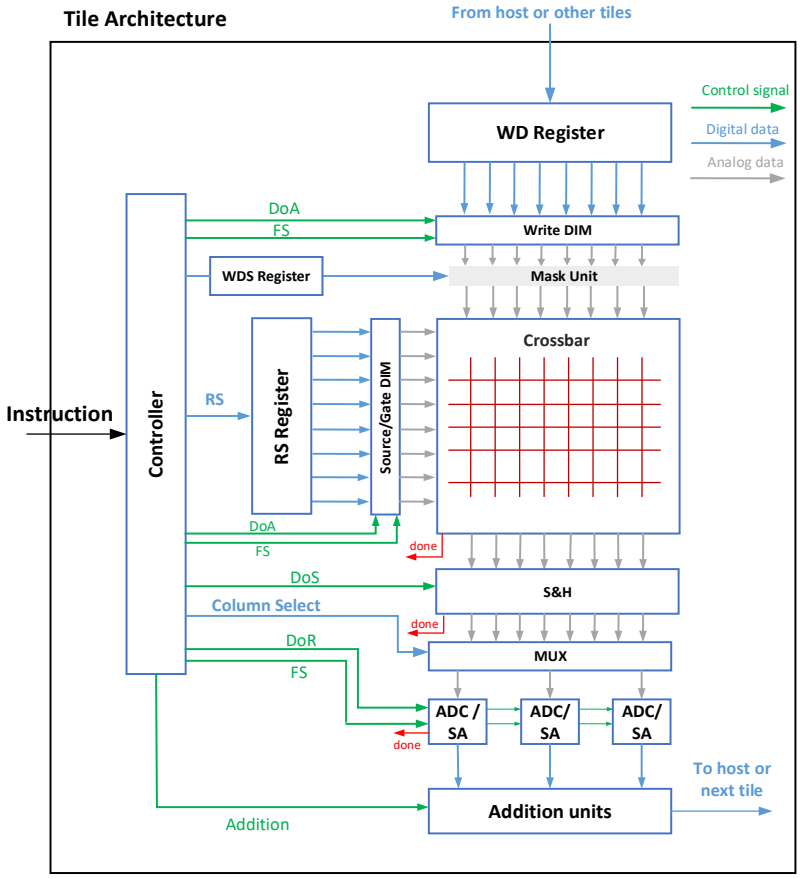

Figure 2: The overall CIM-tile architecture

1) Write operation:

In order to program the crossbar, we need to pass the data and the location where it has to be written into the crossbar. The data itself has to be stored into the Write Data (WD) register whose length depends on the width of the crossbar as well as the number of levels supported by the memristor cells. The information regarding the location of data will be written into the Row Select (RS) and Write Data Select (WDS) registers. The RS register is employed to activate the row and the WDS register indicates the columns in which data has to be written. According to the data stored in these registers, three types of Digital Input Modular (DIM) required to translate digital voltage to the crossbar operating voltage. These drivers are connected to the source and gate of each crossbar access transistor and the third one is connected to the bit-lines of the crossbar.

2) Read/computational operation:

In this category, the operations generate an output and it has to be read by the periphery circuits in the architecture. The generated output can be the outcome of either a normal memory read or computational operation. In contrast to the write operation, there is no need to fill the WD and WDS registers. The RS register again is used for row activation. However, among computation operations, matrix-matrix multiplication (MMM) is a little different than others in the sense that the RS not only has to indicate the active rows, but also can be considered as the data for one of the matrices. When the operation is performed inside the crossbar, the generated analog output has to be captured by the Sample & Hold (S&H) unit. Subsequently, the results are converted to the digital domain using ADCs which are shared among multiple columns.

2.2. Nano-instruction set architecture (nano-ISA)

Table 1: List of nano-instructions

Nano-instruction	Opcode	Immediate	Application
Row Select	RS	Data to fill RS register	Write/Computation
Write Data	WD	Data to fill WD register	Write
Write Data Select	WDS	Data to fill WDS register	Write
Function Select	FS	Function mode	Write/Computation
Do Array	DoA	-	Write/Computation
Do Sample	DoS	-	Computation
Columns Select	CS	Data to fill CS register	Computation
Do Read	DoR	-	Computation

As discussed in Deliverable D4.6, a complex sequence of steps need to be performed in the CIM tile that can be different depending on the (higher-level) CIM tile operation, e.g., read/write, dot-matrix multiplication, Boolean operations, and integer matrix-matrix multiplication. Similar to the concept of microcode, we introduce an instruction-set architecture for our CIM tile that would allow for different schedules for different CIM tile operations. The “Controller" in Figure 2 is responsible for translating these instructions to the actual control signals (highlighted in green). The list of instructions is presented in Table 1. Detail information and description of nan-instruction can be found in Deliverable D4.6.

To translate high-level operations intended for the CIM tile into a sequence of nano-instructions which have to be executed within the CIM tile, we wrote a new compiler, called low-level compiler according to Figure 3. The high-level operations (e.g., MMM) are provided by the high-level compiler which is responsible to search for the operations within the application program that can be performed using the memristor crossbar. The high-level compiler was presented in WP2 and is out of the scope of this report. Based on the requirements or constraints that come from either the tile architecture or technology side, our low-level compiler translates high-level operations to nano-instructions. As depicted in Figure 3, this information is written to the configuration file and passed to the compiler. It is important to note that the sequence of instructions generated by the compiler changes whenever the tile configuration changes. Therefore, by putting this complexity into the compiler, we try to keep the controller as simple as possible. More information about the low-level compiler was presented in Deliverable D4.6. In addition, the content of configuration file and the parameters were defined there, will be explained in the following sections.

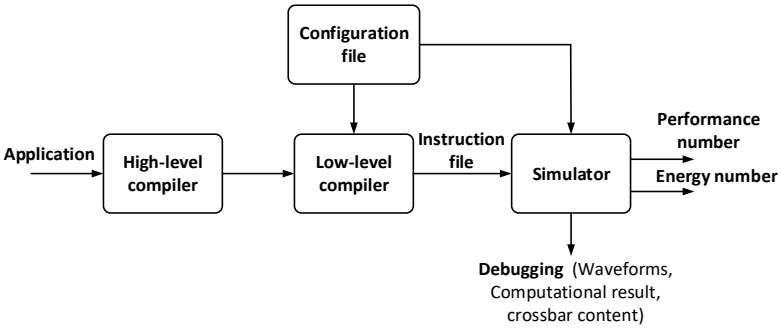

Figure 3: The overall system flow

3. CIM-tile pipelining

The operations in the digital periphery and the analog array can be divided into the following stages: (indicated by different colors in Figure 4)

1. Set up stage (digital): all the control registers (and write data register) are initialized
2. Execution stage (analog): perform the actual operation in the analog array
3. Read out stage (digital): convert the analog results into digital values
4. Addition stage (digital): perform the necessary operations for the integer matrix-matrix multiplication

These stages sequentially follow each other while performing higher level operations translated to a sequence of instructions in our ISA. It should be clear that the pipelining described here is different from the traditional instruction pipelining. In the latter, the latency of each stage should be matched with each other in order to have a balanced pipeline. In the CIM tile, the latency of the operation performed in the analog array is expected to be much longer than the latency of a single clock cycle in the digital periphery. Therefore, it is important that the right signaling is performed between the stages in order to enable pipelining. The introduced execution model to pipeline the operations within the CIM tile, will allow for trade-off investigations between different NVM technologies and the (speed of the) digital periphery.

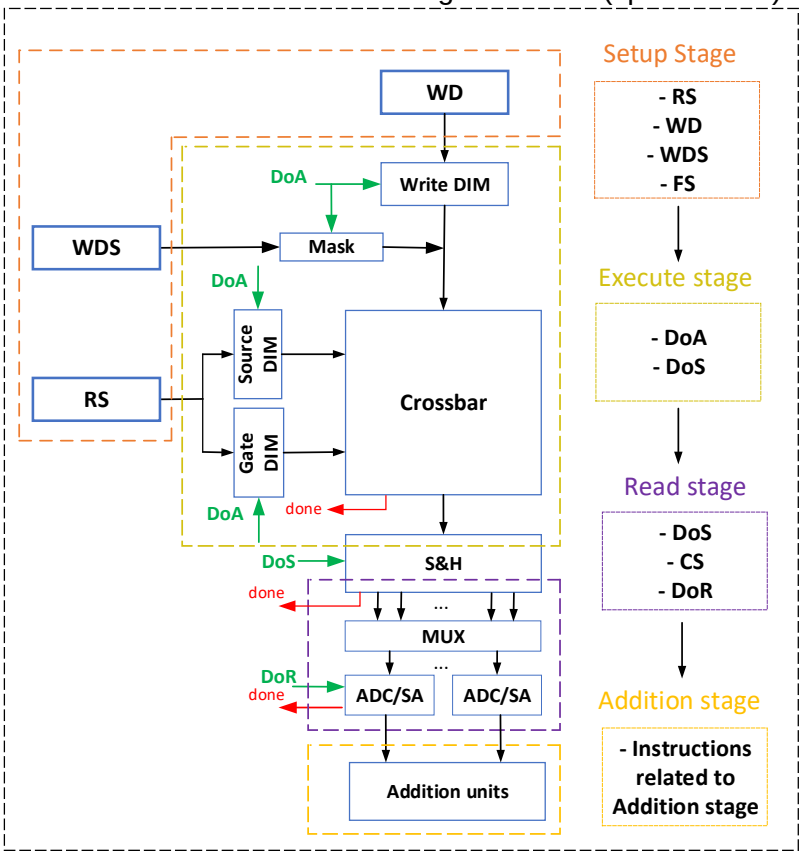

Figure 4: pipelining of the CIM- tile

In contrast to traditional processors where the execution of instructions is split into different stages to enable pipelining, our tile architecture associates different instructions to each tile stage. In the first stage, registers should be filled with new data and the drivers have to be configured. In the second stage, to activate the crossbar using DoA instruction, the operation latency for the previous activation must elapse. The latency of the crossbar, which depends on its technology as well as the operation supposed to be executed, can be captured by either a counter or done signal generated by the circuit itself. Based on the operation which can be

write and read/computational, done signals issued by the crossbar and S&H unit should be used to synchronize the first two stages, respectively. In the third stage, the latency of the Read stage depends not only on the latency of ADCs, but on the number of columns that have to be read as well. Accordingly, groups of columns are read by ADCs sequentially and this stage would be available for the next S&H activation after the last columns already translated to the digital domain.

4. Background on the addition unit

In this section, we briefly review the proposed addition unit (see Figure 4) presented in Deliverable 4.5.

Matrix-matrix multiplication (MMM) is a prevalent operation in many applications that needs processing over numbers representing in a form of one existing datatypes. The presented scheme is supporting integer as a fundamental datatype. To do MMM, the multiplicand has to be mapped to the crossbar. However, (i) since there is a limitation over the number of bits that can be stored in one memristor cell, the elements of multiplicand have to be distributed over several cells. In addition, (ii) due to the constraint on the number of levels that can be supported for the crossbar inputs, in a similar way, each element of multiplier has to be given to the crossbar in several steps. Finally, (iii) in the case that ADCs do not have enough resolution to be able to activate all the required crossbar rows at the same time, we also need to perform it in several steps. According to the aforementioned limitations, in each step, an intermediate result for the MMM is produced. However, in order to get the final result, extra digital processing has to be considered, which is happening inside the addition unit.

The proposed addition unit comprises a maximum of three stages:

1- By giving one bit (or more depends on technology) of the multiplier to the crossbar input, the addition between the outcome of each column (already performed by the crossbar) is done in a way that minimum size of adder and register is required. In the case that the ADC cannot support enough precision, a pre-phase is employed for this stage to first get the final result for each column.
2- Getting the intermediate result for one bit of multiplier in the first step, when the next bit is given to the crossbar, the outcome should be summed up with the intermediate result obtained from previous bit-positions. Similar to the first stage, the addition in this stage is done in a way that minimum size adder and register are employed.
3- Finally, considering the mapping of integer number to the crossbar, if part of a number (just some of the columns representing an entire number) shared with an ADC, the third stage is essential to do the addition between the final outcome of all the previous steps already taken per ADC (addition per ADC).

For detailed information about the addition unit, please read the deliverable 4.5. In this report, we evaluate the proposed structure using our tile simulator in terms of energy and performance.

5. CIM-tile micro-simulator and compiler enhancement

The proposed CIM architecture is generalized making it capable of targeting different technologies with different configurations of the peripheral circuit. The simulator, written in SystemC, models the architecture presented in Section 0 and generates performance and energy numbers by executing applications. The simulator takes as input the program generated by the compiler (presented in Section 2.2 and Deliverable D4.6) which is currently stored as simple (human-readable) text. Besides the program, to simplify design space exploration, the configuration of architecture has to be sent to the simulator via a configuration file in which the user is able to specify many parameters. The simulator produces as output the following: (1) energy and performance numbers, (2) content of the crossbar (over time), (3) waveforms of all control signals, and (4) the computational results. All outputs are written into text files to be used for further evaluation.

The simulator has been written in a modular way, which helps us to easily modify or replace the components shown in the tile architecture with new designs/circuits. In our cycle accurate fully parameterized simulator, each components has its own characteristics like energy, latency, and precision written into the configuration file. Table 2 shows all the parameters that can be set in the file to be used for the early stage design space exploration. Furthermore, the first of its kind feature of our simulator is the ability to calculate energy number according to the data provided by the application. Existing simulators estimate average energy number regardless of data. Our simulator takes into account the data stored in the array to estimate the energy consumption in the crossbar and its drivers. This is achieved by taking into account the data stored in the crossbar cell resistance level, the number of activated rows, and the equivalent resistance of the crossbar. The power consumption of the crossbar and read drivers regarding read/compute operations is given in Equation 4 where R_{rc} is the resistance level of the memristor cell located in row *"r"* and column *"c"*, $P_{DIM_{read}}$ is the read drivers power, and $V(read)$ is the read voltages. In general, R_{rc} and $V(read)$ are members of two sets contain possible resistance and voltage levels, respectively. Furthermore, activation is a binary value that indicates whether row *"r"* is activated and contributes to the power of the crossbar or not. Considering read and compute operations, the summation is performed for the selected rows and all the columns. In addition, for simplicity, the resistance of access transistors, as well as bit-lines, are ignored. The power consumption of write operations is shown in Equation 5 where $P_{DIM_{write}}$ is the write drivers power. $V(write)$ and $I(write)$ are the write voltage and programming current, respectively. In general, $V(write)$ is a member of a set including different write voltage levels. Finally, $activation_c$ determines whether the column *"c"* is activated and would contribute to the crossbar energy or not. The summation is performed over the selected columns in just one activated row. The energy consumption of read/computational as well write operations are shown in Equations 6 and 7, respectively, in which $T_{Xbar(write)}$ as well as $T_{Xbar(read)}$ are the latency of the crossbar for write and read/computational operations. The latency of the crossbar for read/computational operations depends on the peripheral circuits used to capture or read the analog values generated by the crossbar. Using S&H unit to capture the result, its capacitance is charged with different gradient according to the equivalent resistance of the crossbar. Therefore, the result should be captured at the right time when there is a maximum voltage difference on the capacitance of S&H unit for different crossbar equivalent resistances, which helps to be distinguished by the ADC easily. It is worth to mention that the equations are data-dependent and provide the worst-case energy numbers for the crossbar and its drivers.

Table 2 List of parameters used in the configuration file

	crossbar and drivers	**analog peripheries**	**digital peripheries**
Structure	- Number of rows/columns - Cell levels	- Number of ADCs - precision of ADCs	- Clock frequency - datatype size
Energy	- cell write energy - cell read energy - write driver energy - read driver energy	- ADC energy per conversion - SH energy per sample	- energy per adder in addition unit
time	- write latency - read latency	- ADC latency - SH latency	- RS filling cycle - WD filling cycle - WDS filling cycle - CS filling cycle

$$P_{(read,compute)} = \sum_{r=1}^{\#rows} activation_r * \left(\sum_{c=1}^{\#columns} \frac{V^2(read)}{Rrc} + P_{DIM_{read}} \right) \quad (4)$$

$$R_{rc} \in \{L1, L2, \dots, L_n\} \;-\; activation_r \in \{0,1\} \;-\; V(read) \in \{V(r1), V(r2), \dots, V(rn)\}$$

$$P_{(wrute)_r} = \sum_{c=1}^{\#columns} \left(V(write) * I(write) + P_{DIM_{write}}\right) * activation_c \quad (5)$$

$$V(write) \in \{V(w1), V(w2), \dots, V(wn)\}$$

$$E_{(read,compute)} = P_{(read,compute)} * T_{Xbar_{(read,compute)}} \quad (6)$$

$$E_{(write)} = P_{(write)} * T_{Xbar_{(write)}} \quad (7)$$

Due to the execution model and flexibility of our instructions, the simulator is able to add the energy of ADCs which are active during the program execution to the total energy of the tile. Besides the hardware implementation of our digital controller, which can provide an accurate number for this unit, a more advanced model for the energy and performance of the crossbar are our main focus for future work. In addition, thanks to our low-level compiler, the simulator can perform computation with different integer datatype sizes at the same time using the structure proposed in Section 0.

6. Evaluation

The defined CIM tile architecture, ISA, compiler, and simulator allows for various design space explorations. In this section, we will present several of these explorations that are currently possible with our tools. In addition, we evaluate our proposed addition scheme in terms of performance and energy number and compare it with the reference design.

6.1. Simulation setup

Energy and performance model

The values used in our experiments regarding the (technology) parameters are summarized in Table 3. The needed values related to the digital periphery were obtained by using Cadence Genus targeting the standard cell 90nm UMC library. The values related to the three targeted technologies (ReRAM, PCM, and STT-MRAM) were taken from [27] [28] [29] [30] [31] [32]. For all the experiments, we assume the size of the crossbar is 256 by 256 and its input precision. is one bit. The latency of the crossbar is defined from the moment the input voltage is applied and crossbar rows get accessed until the capacitance of S&H is charged. This time also depends on the sensing mechanism and ADC circuitry. In addition, the cycles required to fill the tile registers are computed based on the crossbar size and we assumed the data-buses to be 32 bits wide. The energy and latency values for the ADCs were taken from [33].

Table 3: Value of parameters used for the experiments

Component	Parameters	Spec		
Memristive devices		ReRAM	PCM	STT-MRAM
	Cell levels	2	2	2
	LRS	5K	20K	5K
	HRS	1M	10M	10k
	Read voltage	0.2V	0.2V	0.9V
	Write voltage	2V	1V	1.5V
	Write current	100 uA	300 uA	200 uA
	Read time	10 ns	10 ns	10 ns
	Write time	100 ns	100 ns	60 ns
Crossbar	Structure	1T1R		
	Num. columns	256		
	Num. rows	256		
DIM		Read DIM		Write DIM
	number	256		256
	power	1 mW		1mW
ADC	power	2.6 mW		
	Precision	8 bits		
	Latency	1.2 GSps		
Carry-look ahead Adder		Energy (per computation)		Latency
	8 bits	0.01 pJ		1 ns
	16 bits	0.03 pJ		2.2 ns
	24 bit	0.08 pJ		3.2 ns
	40 bits	0.25 pJ		5.6 ns
	72 bits	0.78 pJ		9.8 ns

Benchmark

As a benchmark, the linear-algebra kernel “GEMM” from the Polybench/C benchmark suite was chosen. In this kernel, first, the multiplicands are written into the crossbar (write operation) and then the actual multiplication (compute operation) is performed. This benchmark was chosen as it intensively utilizes the memory array given that we want to perform DSE targeting different technologies for the memory array.

6.2. Simulation result

In this section, we will present several design-space explorations that are currently possible using our simulator. The insights obtained from these analyses will lead designers to take better decisions for the actual implementation.
In Figure 5, we plotted the (normalized) execution time of running the GEMM benchmark targeting three different technologies for the crossbar array, namely PCM, ReRAM, and STT-MRAM. The simulations were performed assuming a 1 GHz clock frequency for the digital periphery and an 8-bit ADC resolution. We can clearly observe that the number of ADCs greatly impacts the execution time. By adding more ADCs, the total execution time can be reduced as the cycles needed to read out the data from the crossbar array can be reduced. Although STT-MRAM has faster write time, due to the less number of write operations to program the crossbar compared to the computational operations, the improvement on the execution time is negligible. An interesting observation is that the performance does not improve much when moving from 32 to 64 ADCs. This can be explained by the fact that at some point, the latency of the read stage is no longer dominant and further reducing the readout time has little impact on the total execution time. Finally, regardless of the number of ADCs, the energy consumption is almost constant (small fluctuation due to the data randomness) since the number of conversions is always fixed.

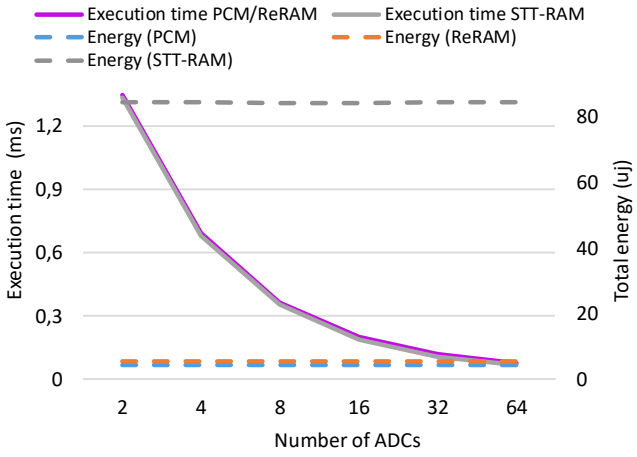

Figure 5: The impact of number of ADCs on execution time of GEMM benchmark

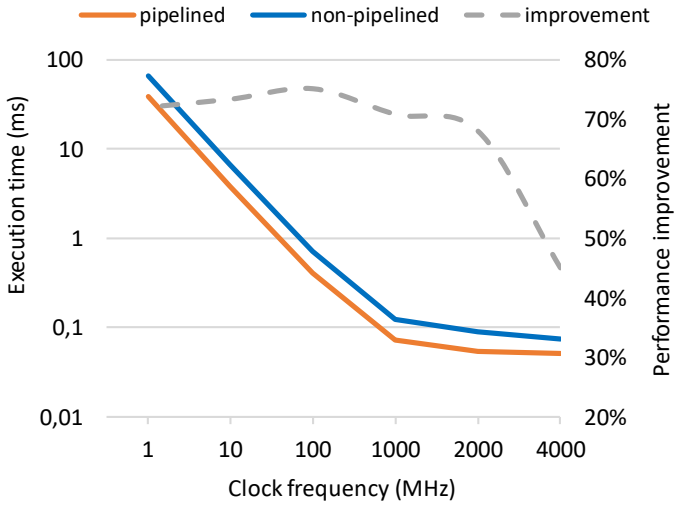

Figure 6: Performance improvement due to the unbalanced pipelining of tile used ReRAM/PCM device for GEMM benchmark

The latency of the operations in the (analog) crossbar array is a constant number. Therefore, it is interesting to determine how fast the digital periphery should be clocked in order to 'match' this latency in order to make the pipeline more balanced. In the following investigation, we have fixed the number of ADCs to 16 and ran the GEMM benchmark at different frequencies. Figure 6 clearly shows that performance improvements can be gained by raising the frequency of the digital periphery. However, increasing the clock frequency beyond 1 GHz does not result in much better execution times as the analog circuits (relatively) are becoming the bottleneck. This DSE allows a designer to make the different stages of the tile more balanced. A positive side-effect is that pipelining more balanced stages will usually lead to better performance improvements over an non-pipelined design.

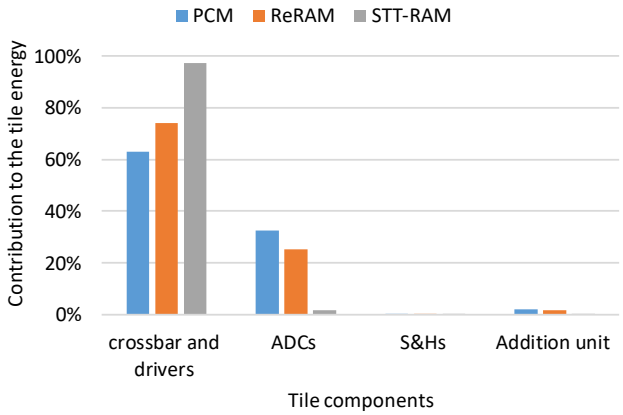

Figure 7: Contribution of different components to the energy consumption for GEMM benchmark

Figure 7 depicts the relative energy spent in the different modules when running the GEMM benchmark for 16 ADCs and using an 8-bit datatype. We can clearly observe that the largest energy consumer is still the crossbar and its drivers. In the PCM case, the relative energy consumption of the ADCs and crossbar are close to each other (compared to other technologies). The reason for this is that the power consumption of PCM is relatively lower

compared to the ReRAM technology due to the higher cell resistance (see Table 3). This in turn increases the relative energy consumption of the ADCs.

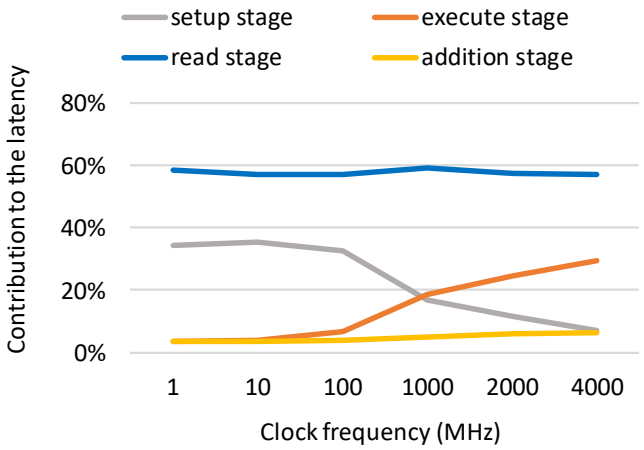

Figure 8: Contribution of each pipeline stage on the latency of the tile considering different clock frequencies

Figure 8 depicts the relative time that the GEMM application spends in each of the 4 stages plotted against the frequency of the digital periphery. We can clearly observe that with a low frequency, the read and setup stage are completely dominant in the total latency. By increasing the clock frequency to 10MHz, the latency of the setup and read stages reduces. Still, their relative contribution remains unchanged. As the clock frequency is increased more, the latency of the analog components starts to rise (relatively). In addition, the relative contribution of the read stage to the total latency is almost fixed. Since many columns share an ADC, the read stage, which is composed of analog (latency of ADC) and digital (decoding latency) latency, inherently imposes much latency regardless of clock frequency. This information can be used to determine the number of pipeline stages for the actual implementation.

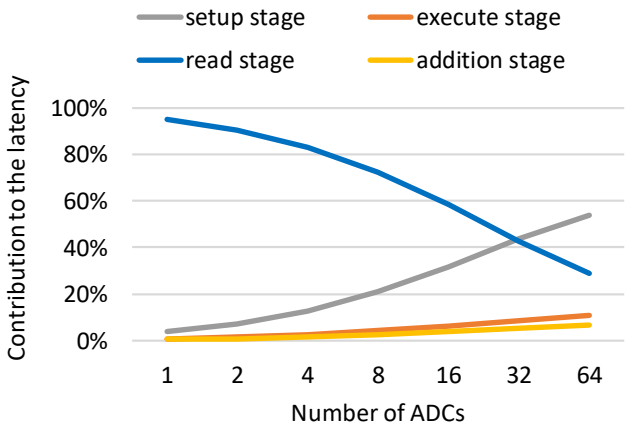

Figure 9: Effect of number of ADCs on the latency of pipeline stages in 100 MHz clock frequency

Figure 9 depicts the relative time that the GEMM application spends in each of the 4 stages plotted against the number of utilized ADCs. It should be clear that increasing number of ADCs, the number cycles spend in the read out stage is greatly reduced. Consequently, we

can observe that the relative contribution of the setup stage to the total latency grows accordingly. The contribution of the other stages to the total latency is almost negligible.

Proposed addition scheme

In the following, we will evaluate our proposed addition scheme compared to the reference design. In the reference design, we assume that a single adder to perform accumulation between shared columns and different bit positions of the multiplier are connected to each ADC. The size of the adder is fixed and must be chosen based on the largest possible value resulting from the MMM - it is specified in Equation 8. This means that the value produced by the ADC is merely an intermediate result that must be summed up into the accumulator – remember that only a single bit of the multiplier is multiplied with the multiplicand and each ADC read-out correspond only to a single bit-position of the multiplicand. Due to the previously stated manner of summation, the intermediate results must be shifted by the correct number of positions (based on the bit-positions of the multiplier and the multiplicand) before entering the adder.

$$Adder\ size\ = \\ int\ size(multiplier)\ +\ int\ size(multiplicand) + log2(crossbar\ height) \tag{8}$$

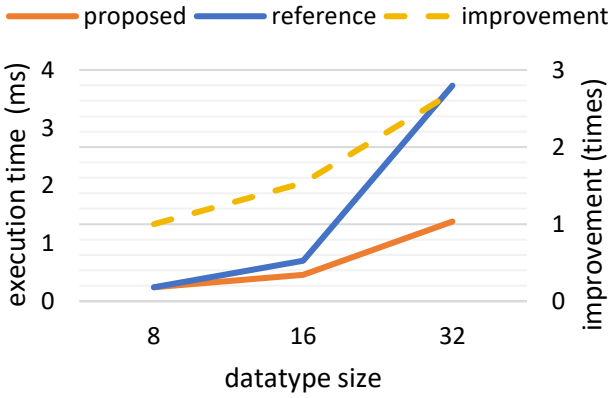

Figure 10: Execution time for different integer datatype sizes

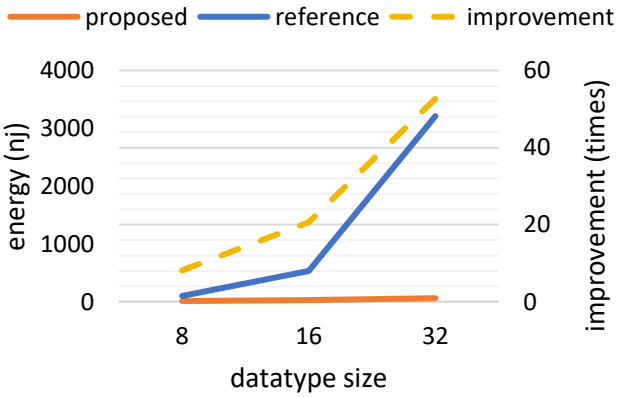

Figure 11: Energy consumption of addition unit for different datatype sizes

Reading data from the crossbar's columns (read out phase) is inherently slow mainly because of the shared ADC between multiple columns and can be considered as the bottleneck of the architecture. However, as the size of adder grows, its latency can be dominant over the latency imposed by ADC. Considering the proposed design in our experiment, since the crossbar has 256 rows, the maximum size of the first two adders is always 8-bit regardless of datatype size. Therefore, there is no extra overhead on the latency of read out phase. Rather, in the reference design and according to the Equation 6, the required size of adder is much larger. Accordingly, considering a 1 ns latency for the ADC, as the size of adder increased more than 8-bit, it becomes the bottleneck of the system and makes the read out phase more costly.

Figure 10 depicts the execution time of the kernel (including writing to the crossbar). In this experiment, we assume that the size of the integer numbers matches the number of columns shared by one ADC. According to the figure, the size of the first two adders for the proposed design are always constant (here 8-bit). Rather, as the datatype size increases, a larger adder has to be employed for the reference design. Considering an 8-bit datatype size in Figure 10, a 24-bit adder has to be used for the reference design, which imposes a 3 times bigger latency than an ADC. However, due to the abundance of ADCs, the entire readout phase is not the bottleneck of the system (see Table 3: Value of parameters used for the experiments for the crossbar latency). Therefore, there is no performance improvement at this point. Figure 11 shows the energy improvement achieved by the proposed design. Although the number of computations is always the same, they are performed with smaller adders, which has a quite good impact on the energy consumption of the addition unit. Finally, the impact of the number of ADCs on the execution time and energy of the addition unit using 32-bit datatype size are presented in Figure 12. As the number of ADCs increases, the performance is improved. In addition, although more adders are employed, their size is decreased, which leads to less energy consumption.

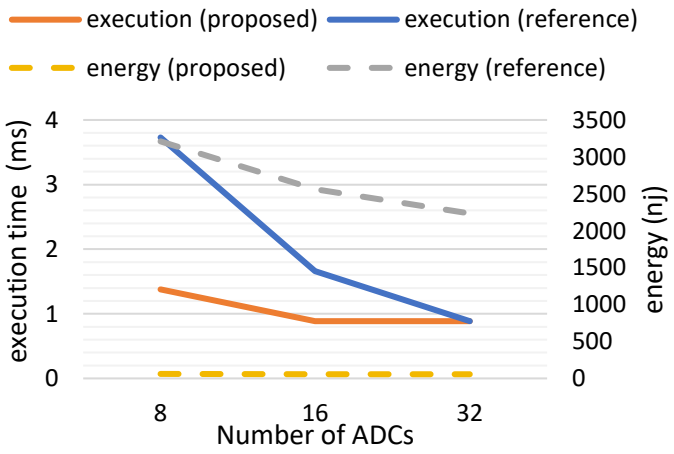

Figure 12: Energy consumption of addition unit and execution time of the benchmark for different number of ADCs

7. IMEC nano-simulator for CIM-A/P tiles

7.1. IMEC nano-simulator for CIM-A/P instantiated for STT-MRAM technology

Making an accurate hardware aware simulator for CIM block is an important goal of MNEMOSENE. Hardware aware simulator provides valuable insight for architectural exploration in different abstraction levels. In imec, our focus is close to technology operations. Therefore, we worked on a nano-simulator which abstracts the functionality of STT-MRAM memory cells along with its peripherals. Figure 13 illustrates the different hierarchical levels in the MNEMOSENE architecture template as defined in WP3. It also shows the position of the imec nano simulator in the global simulation platform. In particular, it directly interfaces with the micro-architecture simulator which is developed in WP3. The latter abstracts the functionality up to number of clock cycles required for read and write, total area, total energy for reads and writes for different memory dimensions. The IMEC nano-simulator provides these numbers in a parametrized way.

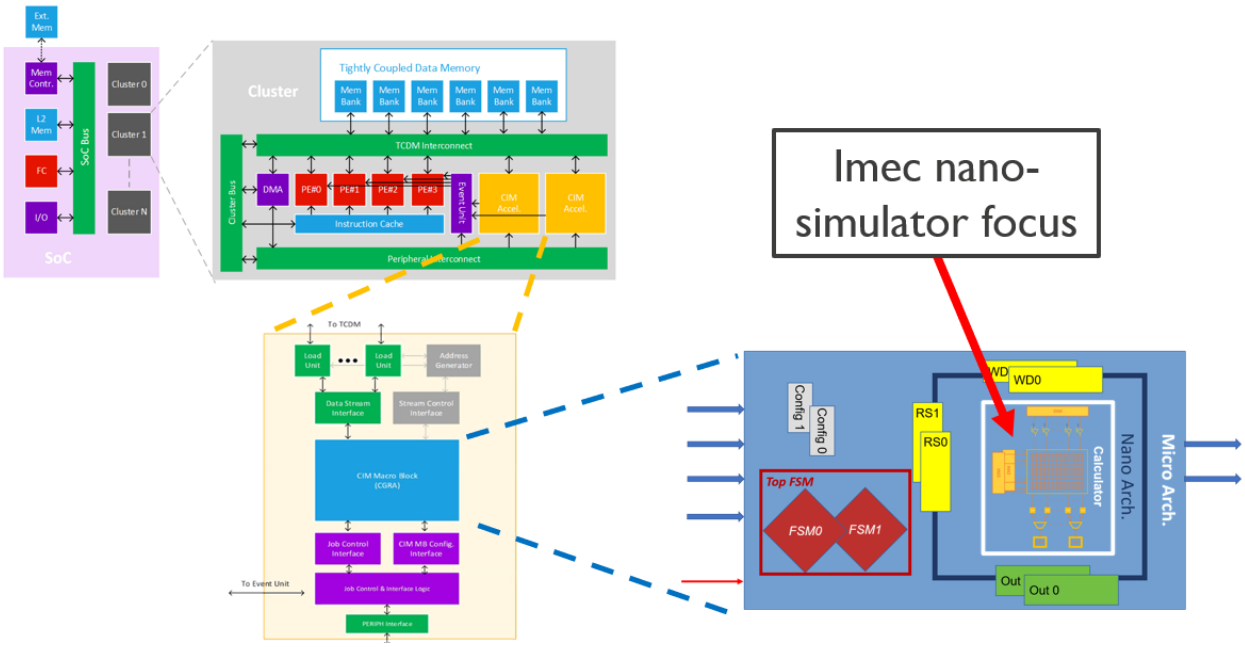

Figure 13: A complete overview of MNEMOSENE simulator platform and position of the imec nano-simulator

The Imec nano simulator mainly wraps the behaviour of the memory cells, sense amplifiers, address decoder, and row/column driver lines. To illustrate the functionality with a real memory technology and to provide quantitative numbers for delay, energy and area we have instantiated it for the realistic STT-MRAM macro which has been developed at IMEC for scaled technology nodes [refs]. All the important components are parametrized and the delay, energy and area have been calibrated based on measurement of fabricated test structures.

Our Imec nano-simulator is accurate and at the same time faster than low-level circuit simulations. Additionally, to be able to provide it to all the other partners, the simulator should be packed like a black-box to not reveal protected confidential information about the IMEC

memory technology. This new simulator is written in python and converted to a secure executable file to run independently without any third-party tools.

Using this simulator, it is possible to simulate the STT-MRAM cells plus the peripherals to explore metrics like energy consumption and latency for any application. It is possible to add other relevant metrics if it is required.

For simulations of the STT-MRAM cells and the sense amplifiers, we extract the data and equations from the analog simulations which were previously reported in D4.5. Currently, those data are limited to read/write instructions. We can add energy/latency data for CIM-A instructions (like in-memory binary operations) when the CIM-A simulation results are available.

In addition to STT-MRAM cells and sense amplifiers, the imec nano-sim includes a model for memory peripherals like address decoder and drivers. Specifically, we have modelled a version of our Address Calculation Accelerator (ACA). This unit performs address processing inside memory which will be explained in detail in the following sections.

Figure 14 shows the input/output files of the black-box nano-simulator. Table 4 explains the input/out files.

Table 4: input/outputs files of nano-simulator

Instruction file	Input	Contains the instruction list (with/without ACA extension) to the memory
Configuration file	Input	Contains the configurations of the STT-MRAM memory, the path to the memory image file, instruction file, and result file
Memory image file	In/Out	Stores the content of the memory before, after, and during the simulation.
Results file	Output	Contains the detailed results of the simulation, including energy/power/time consumption of each element in the simulation

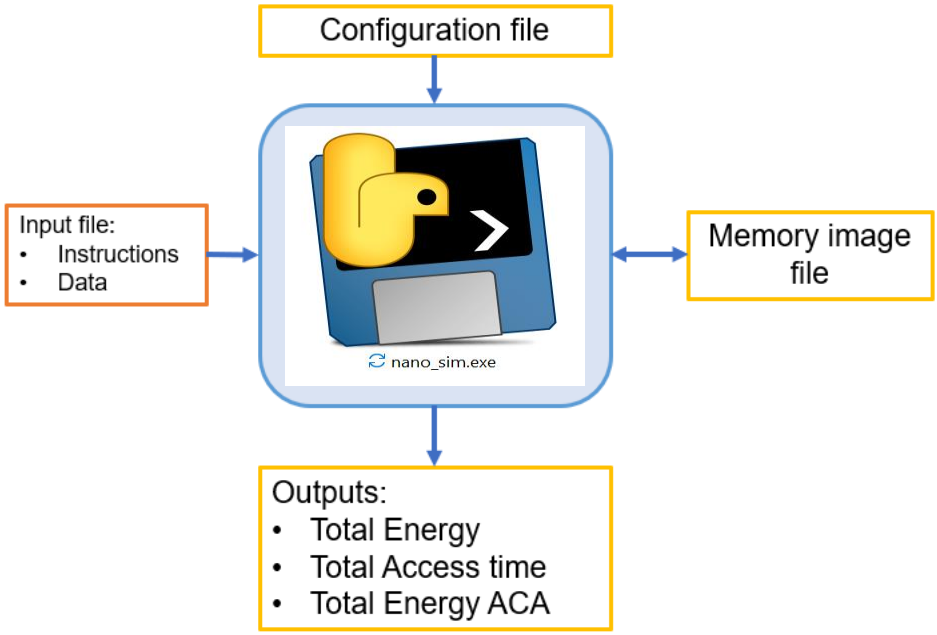

Figure 14 Input/Output files of the black-box nano-simulator

Figure 15 shows an example of the configuration file. Here the user should define the configurations of the STT-MRAM core as well as the locations to load/store other interfacing files. The initial state of memory affects the switching power/latency. Therefore, it is possible to always start from a zero state by activating the init_boolian in the configuration file.

```
#Memory configurations
number of arows(R),32
number of columns(C),96
number of data bits(D),32
initialize memory image with zeros(init_boolian),1
address of memory image(memory_file),32x32\data.csv
address of instruction file(inst_file),32x32\memtrace_out_ver_1.csv
address of result file(result_file),32x32\results_memtrace_out_ver_1.csv

```

Figure 15 An example of the configuration file

```
READ 'read_address'
WRITE 'write address','write data'
ACAR 'row_start array','row_inc','row_cycles','col_start array','col_inc','col_cycles','ins_repeat'
ACAW 'row_start array','row_inc','row_cycles','col_start array','col_inc','col_cycles','ins_repeat', 'write_data array'
```

Figure 16 template of the instruction file

Figure 16 shows the template of the instruction files with currently supported instructions. Every instruction comes with a set of operands. For Read instruction, it is only the address. For write instruction, it is the address and the write data. Nano-sim also supports ACA instructions for burst read (ACAR) and write (ACAW). We will discuss its operands later in this document.

We have used some constants parameters in the nano-sim which is defined by the STT_MRAM technology. Below are those parameters:

```
#Memory configurations constants (fit for imec STT_MRAM)
G= 16          #bit slice multiplexer factor
```

```
N= 2            #number of sub-arrays
if(D>64):  N=4 #number of sub-arrays
P= 0.75         #duty cycle (pulse high time)
T= 26.71        #clock period in [ns]

#Global constants
Fr = 37.443834e6;   #reference clock frequency
Pr = 0.75;          #reference clock duty cycle
```

The energy and access time consumed by each operation should be embedded inside the simulator by using the provided equations. The following subsections discuss the main supported operations in nano-sim.

7.1.1. Memory read

Read energy is mostly dominated by the sense amplifiers. Static power consumption during reading is calculated with Equation 1:

Equation 1

$$Pr_{stc} = \frac{P0 \times (W + W0) \times (D + D0 \times N) \times F0}{(F0 + Fr)}$$

Dynamic read energy consumption per each word is calculated with Equation 2:

Equation 2

$$Er_{dyn} = \frac{P0 \times (W + W0) \times (D + D0 \times N)}{(F0 + Fr)}$$

Read access time is calculated with Equation 3:

Equation 3

$$Tr = Ta + Tb \times (W^{aa})$$

Where the parameters which are used in these equations are as follow:

```
Ta = 2.93e-9
Tb = 0.06e-9
aa = 0.83
P0 = 0.526e-6
W0 = 17.00
D0 = 9.741
F0 = 5.78e6
```

7.1.2. Memory Write

STT_MRAM cells consume energy to change their internal state. Therefore write energy is expected to be more than read-energy. However, when the writing of data does not include a change of state (for example writing ‘1’ into a memory cell which is already ‘1’), the power consumption will be limited to drive the bit-lines.

In conclusion, write energy should be calculated differently in these 4 scenarios:

1- Writing '1' in a cell which is already '1' (non-flipping 11)
2- Writing '1' in a cell which is already '0' (flipping 10)
3- Writing '0' in a cell which is already '1' (flipping 01)
4- Writing '0' in a cell which is already '0' (non-flipping 00)

Static write power in nano-sim is calculated with Equation 4

Equation 4

$$Pw_{stc} = \frac{(n_{01} + n_{11}) \times P0_1 \times (W + W0_1) \times (D + D0_1 \times N) \times F0_1}{(F0_1 + Fr)} + \frac{(n_{10} + n_{00}) \times P0_0 \times (W + W0_1) \times (D + D0_1 \times N) \times F0_1}{(F0_1 + Fr)}$$

Static power is the same for flipping or non-flipping writes. However, static power is only consumed when the clock signal is high. As the clock duty cycle is a parameter of the memory, we adjust the static power as shown in Equation 5

Equation 5

$$Pw_{stc} \leftarrow Pw_{stc} * P/Pr$$

Dynamic energy consumption is calculated with Equation 6:

Equation 6

$$Ew\_dyn = \frac{(n_{01} + NFF \times n_{11}) \times P0_1 \times (W + W0_1) \times (D + D0_1 \times N)}{(F0_1 + Fr)} + \frac{(n_{10} + NFF \times n_{00}) \times P0_1 \times (W + W0_1) \times (D + D0_1 \times N)}{(F0_1 + Fr)}$$

The write time changes based on the value which is written in the cell.

Equation 7

$$Tw_1 = (Ta_1 + Tb_1 \times (W^{aa_1}))$$

Equation 8

$$Tw_0 = (Ta_0 + Tb_0 \times (W^{aa_1}))$$

Equation 7 calculates the write time when writing '1' into the cell while Equation 8 calculates the write time for writing '0'. As the write instruction is a word-level instruction, the word write time is defined by the maximum write time of all the cells.

Followings are the parameters used in these equations:

```
#For anti-parallel to parallel switch (1 -> 0)
Ta_0 = 3.76e-9
Tb_0 = 0.0016e-9
aa_0 = 1.76
P0_0 = 1.259e-6
W0_0 = 66.44
D0_0 = 1.729
```

```
F0_0 = 197e6
#For parallel to anti-parallel switch (0 -> 1)
Ta_1 = 5.59e-9
Tb_1 = 4.08e-9
aa_1 = 0.0
P0_1 = 1.628e-6
W0_1 = 45.94
D0_1 = 2.001
F0_1 = 99e6
#General parameters
NFF = 0.5 #non-flipping factor
n_00=0.0  #number of bit writes from 0 to 0
n_01=0.0  #number of bit writes from 0 to 1
n_10=0.0  #number of bit writes from 1 to 0
n_11=0.0  #number of bit writes from 1 to 1
```

Total energy consumption is calculated by the summation of dynamic energy with static energy (static power multiply by run time).

Whenever a memory write happens, nano-sim updates the memory image file. Therefore we keep track of all the changes during simulation. Memory image file is a CSV file where every line of it contains the data for every row of the physical memory.

7.1.3. BUS energy

Even though energy consumption over the memory bus is not part of the nano-sim scope as shown in **Error! Reference source not found.**, we have added the option to include the BUS energy in the results. This is because we wanted to show how much our address calculation accelerator improves energy efficiency. Using the experimental results in [34], we concluded that in 65nm technology with 150MHz clock frequency, every bit-transfer in the TCDM BUS consumes around 0.18pJ. This number is used in our simulations.

7.1.4. CIM-P Address Calculation Accelerator

ACA is an additional unit that can be used instead of a conventional address decoder. The idea is rather than calculating the access word address in the processor and sending a series of addresses to the memory, this process happens inside the memory. In section 7.2 we explained ACA in more detail.

ACA is a fully digital circuit and mainly made of a control unit (state machine) and two shift registers, one with the size of the number of rows (R) and another one with the size of the number of columns (C), as illustrated in Figure 17. To include the ACA in our nano-sim, we measured the power consumption of the logic block with digital circuit simulations.

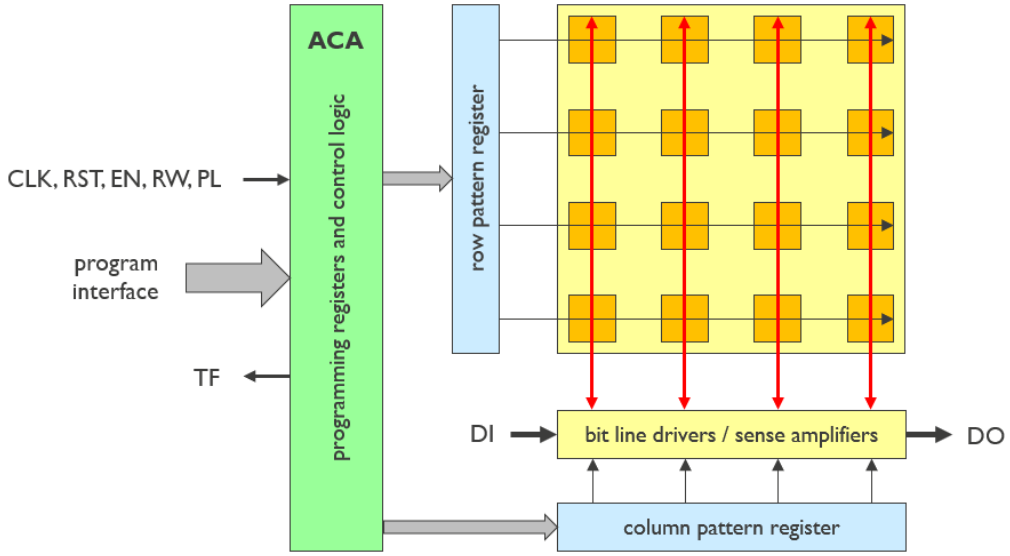

Figure 17 ACA circuit block diagram and position in the full memory macro

As the dominant source of power consumption is the shift registers, we used **Error! Reference source not found.**, Equation 10, and Equation 11 to calculate the static power, energy to load the registers, and energy to shift the registers.

Equation 9 Static power

$$P_{stc} = 2.0 \times 10^{-9} \times (R + C)$$

Equation 10 Load Energy

$$E_{ld} = 2.0 \times 10^{-15} \times ((R \times Ld\_R) + (C \times Ld\_C))$$

Equation 11 Shift energy

$$E_{sht} = 3.0 \times 10^{-15} \times ((R \times Sh\_R) + (C \times Sh\_C))$$

Where Ld_R and Ld_C are active for every register load instruction and Sh_R and Sh_C are active for every register shift instruction. It worth mentioning that only the E_{sht} is dedicated for ACA while P_{stc} and E_{ld} is also consumed when a normal address decoder is in place.

Figure 18 shows an example of the output file from the nano-sim. It includes relevant information like the energy and access time of each component.

```
32x32\memtrace_out_hor_0_cfg_pass2.csv
memory configuration: R=32 C=384 D=32
########RESULTS############
Total_Energy=3530.266nJ
Total_Memory_Energy=2455.777nJ
Total_BUS_Energy=898.637nJ
Total_ACA_Energy=175.852nJ
Total_Time=550.75uS
Avg_Power=6.41mW (total_energy/total_time)
##########################
```

Figure 18 An example of the result file (down)

7.1.5. CIM-A Accelerators

It is possible to include the information about the CIM-A operations into the nano-sim. In that case, the read and write of the data has to be replaced by a modified read and write where also logic or arithmetic operations are incorporated. For our nano-simulator, this simply means including a wider set of memory operations in the list, and to add the corresponding delay, area and energy results on top of already available read and write operations.

Due to time limitations, we have not yet performed this integration in the MNEMOSENE scope so we cannot show quantitative results at this stage.

7.2. Optimized CIMP-tile for address calculation

One of the main goals in Mnemosyne is to reduce the data transfer between the memory and the processor cores. An important part of transferred information is the memory word address which the processor wants to access. Normally each packet of data that moves between a processor and a memory in a general processor SoC (Figure 19) contains 3 important parts:

Instruction	Target word address	Operands

For example, in a conventional system to write in a line of memory, we provide the target word address to be written. Then instruction is the “write instruction” and the operand is the “write data”. When performing an in-memory process, it is possible to give higher-level instructions to the memory block. For example, an instruction can include a binary AND between the Operand and the content of the target word address.

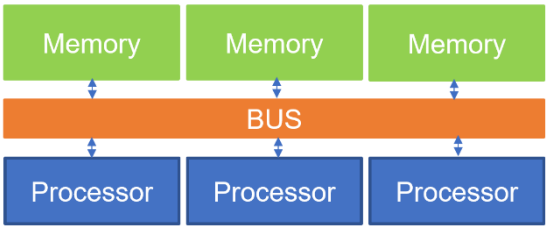

Figure 19 A general processing SoC

The overhead of the “target address word” in every transaction can be considerable (especially for low word resolutions which is a new fashion in edge applications). Address bits scale up with the number of words in the memory and adds considerable overhead to each memory transaction. For example, to access an 8-bit word in a relatively small 1MB memory, it is required to transmit an address with 20 bits.

For the important domain of streaming applications, it is always required to access regular or semi-regular repetitive accesses to one- or more-dimensional arrays and other composite data types. This will be translated at the memory hardware-level into a stream of consecutive addresses in the memory (which is called burst access for the full regular situation). In this case, it is much more efficient to just send the first address and the number of accesses in the burst and to calculate the explicit address locally in the memory. This optimization can easily make the processor-memory communication in terms of address and control commands negligible, with savings up to a factor 100 (as we will show in the results). But it requires a disruptive hardware modification in the periphery of the memory macro. The basic hardware concepts have been reported in the earlier deliverable D4.4.

To generalize this concept, an application may require a semi-regular access pattern to the memory which is not necessarily consecutive burst access. For example, as it is illustrated in Figure 20, applying a 3x3 kernel on a 2D image which is mapped linearly in the memory requires reading 3 separated sections of the memory. This is not fully regular any longer because especially at the image boundaries the repetition is disturbed. Moreover, in many streaming applications, the neighbourhood to be extracted from the full image (or array in general) is not fully dense and holes are present in the pattern. We also want to support such semi-regular but still repetitive patterns.

As described in detail in previous deliverables (D4.4 and D4.6), we have come up with an optimized circuit scheme (an imec IP) to reduce the number of transactions required for address transfer by the implementation of a hardware-accelerated logic block to generate a complex pattern of addresses locally inside the memory. Besides, we have started with the implementation of a modelling and simulation framework to support such CIM-P modifications at the nano-simulation level.

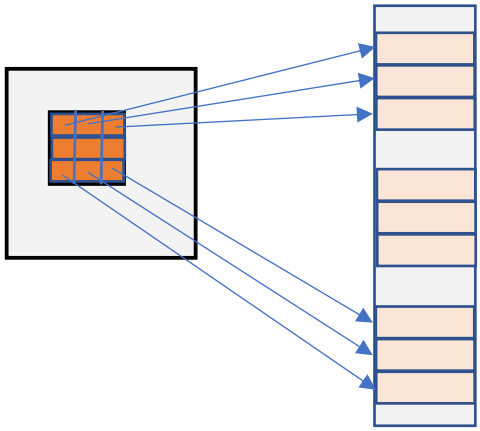

Figure 20 Access memory pattern for a 2D convolution. Apply a 2D kernel in a 2D image (Left). Equivalent access pattern for 1D mapping in the memory (right)

Our Address Calculation Accelerator (ACA) contains two shift registers (row and column registers) and control logic as it is shown in Figure 21. The control logic block (FSM) understands the packed ACA instructions and unpacks them. ACA can generate sequences both in rows and columns when multiple words are stored in one row. Additionally, it is possible to select only part of a word (like masking) when required as shown in Figure 22.

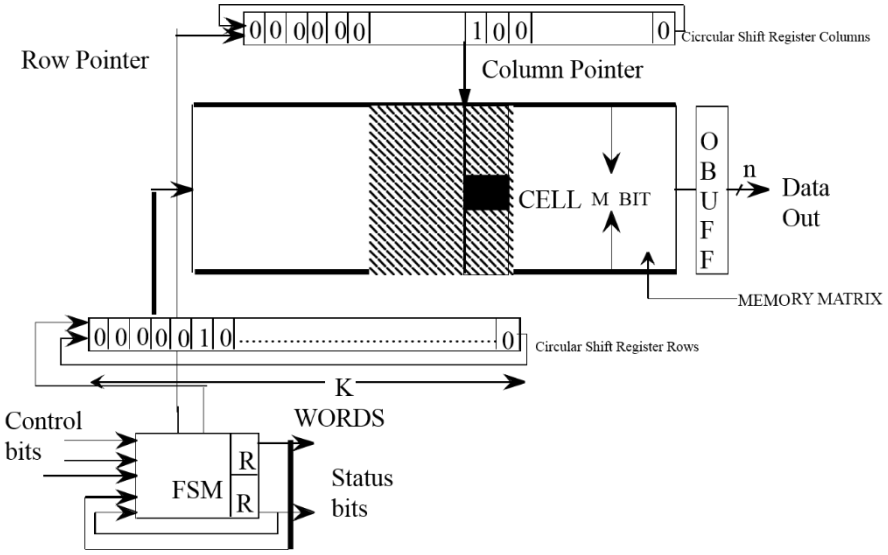

Figure 21 Address Calculation Accelerator block

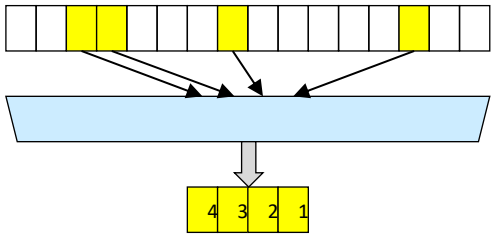

Figure 22 Partial selection of a row/colunm in ACA

To use ACA, the processor should pack several memory access patterns in a form of an ACA instruction. We assume this is happening offline during compile time. In this case, the compiler is aware of ACA instructions. Therefore there is no run-time process required to pack the memory accesses.

When using ACA, many small transactions can be packed in the following format:

Instruction	ACA Operands	Operands

ACA operands are used by the ACA unit to unpack the sequence of addresses. This means the instructions compiled in the processor should be packed using an ACA aware compiler. The following table lists the operands that are used for ACA:

row_start(s)	The start position(s) of the row
row_inc	The amount of increment on the row in every step
row_cycles	The number of shift cycles for the row shift-register
col_start(s)	The start position(s) of the column
col_inc	The amount of increment on the column in every step
col_cycles	The number of shift cycles for the column shift-register
ins_repeat	The number of reaping this instruction

In the current implementation, we have a single start position for row and column. When selecting several row/columns, it is required to have several row_start / col_start operands. It is also important that the memory core can accept such a configuration. For example, to perform in-memory binary operations between two rows of the memory, we should have two row_start active bits.

We are aware that a single cycle shift registers can be expensive to implement for a larger scale. Therefore, it is possible to restrict the row_inc/col_inc numbers to simplify the hardware. Additionally, As the speed of digital peripheral normally is faster than the memory access time, it is possible to perform the shift operation in several digital clock cycles. For example, if the digital clock frequency is 1GHz and memory access time is 10ns, shifting the active bit in the shift register can take 10 clock cycles before the memory is ready for the next access.

7.3. CIM-P ACA compiler

When using memory with in-memory process capability (for example in a platform same as Figure 13), the processor can outsource some part of the computation to the CIM block. In this case, the CIM block accepts higher-level instructions. To perform this kind of computation and processor-CIM communication, the program compiler of the processor needs to be aware of the CIM features.

As we introduced the ACA logic block in the CIM, we also needed to compile the application with an ACA aware compiler. Rather than modifying the existing compilers, we have made a separate ACA compiler that operates after a conventional compiler. The responsibility of the ACA compiler is to detect the access patterns to the memory and packed them by using the ACA instructions. The current version of the ACA compiler is performing a simple search. Therefore, it may be slow for big applications and it may miss some of the more complex patterns. Further optimization of this compiler should be done in future work.

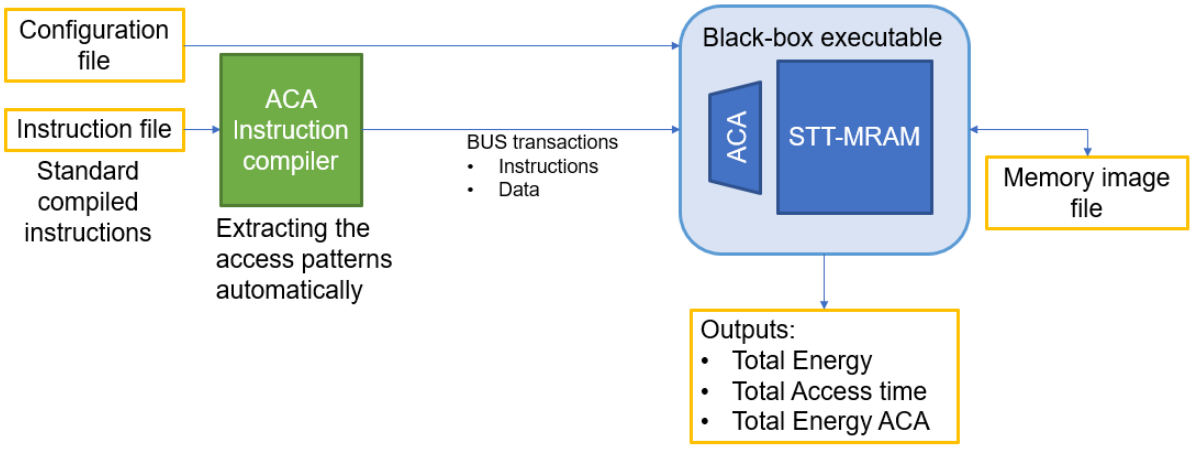

Figure 23 The flow of using ACA compiler and black-box nano-simulator

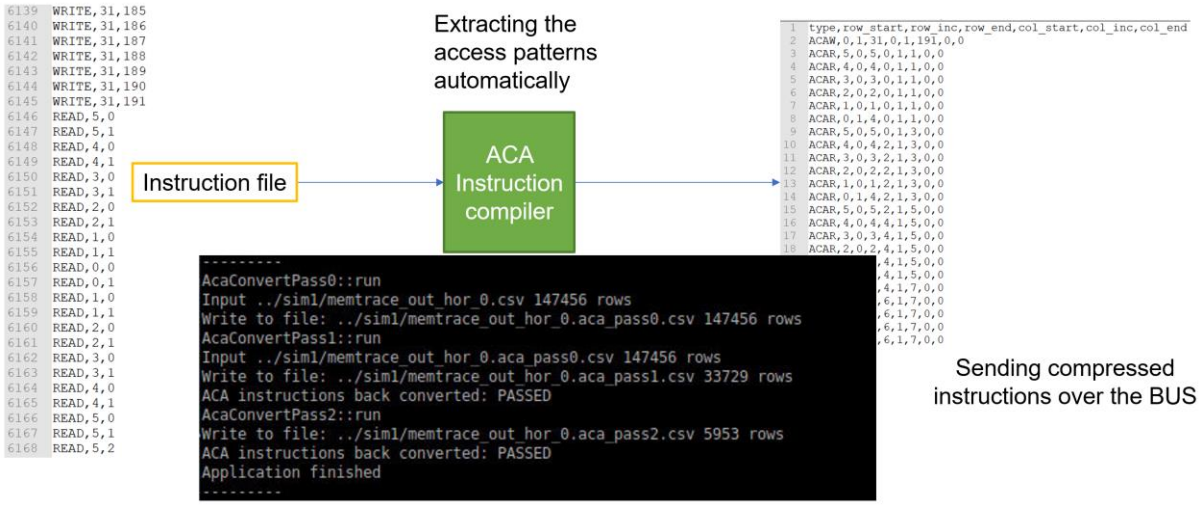

Figure 24 A simple example of input and outputs of the ACA compiler

Figure 23 shows the flow of using the ACA compiler and the nano-sim. ACA compiler compresses the instructions which require memory access with a specific pattern. Figure 24 shows an example of the input and output of the ACA compiler. In this example, we only use read/write instructions but ACA is not limited to these instructions. ACA compiler only searches for memory access patterns and does not interfere with the instruction which is supposed to be executed inside the memory.

7.4. Results from application case studies for CIM-P

In this section, we show some of the results of our experiments by using our IMEC black-box nano-simulator instantiated for the ACA compiler.

7.4.1. Synthetic application case

As the first experiment, we tried to integrate this nano simulator with the TUe micro-simulator for a very small synthetic application. Following is the result.

Integration with TUe simulator:

STT_MRAM

- Total energy = 181 nJ
- Total time = 45 µS

ACA

- ACA energy = 1.01 nJ
- Instructions compression ratio over the BUS = 518X

7.4.2. Basic implementation of guided filter application

To obtain a realistic and representative case study from the streaming data and signal processing domain, we have collaborated with the IPI group of Prof. Wilfried Philips and Prof.

Bart Goossens at UGent & IMEC, Belgium. They work on advanced image processing algorithms and together we have chosen a representative image processing technique called “guided image filtering” [35]. This application uses two images as input and guide and performs repetitive operations on the input image using the guided filter as shown in Figure 25. In our case, the sizes of the input image, guided filter, and output are the same.

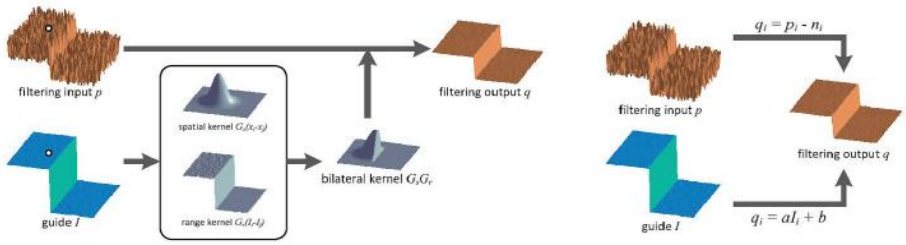

Figure 25 Guided image filtering application [36]

This application follows the pipeline shown in Figure 26 to process an input image.

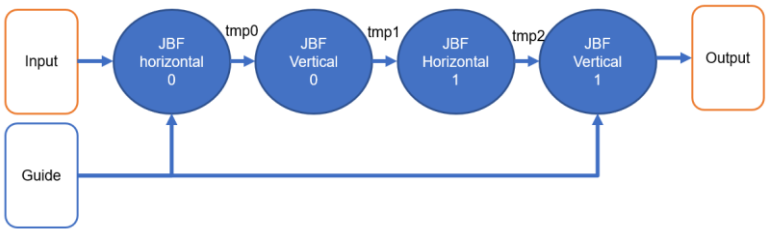

Figure 26 The pipeline of the guided image filtering application. JBF stands for “Joint Box Filter”

As mentioned before, ACA can reduce the number of individual transactions over the BUS by packing/unpacking the addresses. In these experiments, we measured the number of individual transactions before and after using ACA compression. Please note that we only compress the address/instruction fields and operands are required to be transferred in the packet without any compression. Additionally, we run the experiments for different input sizes, as it affects the compression ratio.

Table 5 shows the results of using the nano-simulator with ACA compression for different image sizes in the guided filter application. Also, note that the simulation results also depend on the initial memory state (for example input image in this case) as the energy/time consumption is different when switching from ‘1’ to ‘0’ and from ‘0’ to ‘1’ in an STT-MRAM memory cell.

Table 5 Results of implementation of guided image filtering in nano-sim

Image size: 32×32						
Kernel	The normal number of access	Compresses number of access (ACA)	Compression Ratio	STT-MRAM Energy (nJ)	ACA Energy (nJ)	Total time (µs)
Input	33793	5953	18%	496	12	109
Guide	36865	5954	16%	289	13	118
Tmp0 (Hor0)	147457	5954	4.0%	2456	176	551
Tmp1 (Ver0)	73729	5954	8.0%	1228	46	275
Tmp2 (Hor1)	73729	5954	8.0%	1224	46	275
Output (Ver1)	3073	1	0.03%	118	1	30
Total	**368646**	**29770**	**8%**	**5811**	**294**	**1358**
Image size: 256×256						
Kernel	The normal number of access	Compresses number of access (ACA)	Compression Ratio	STT-MRAM Energy (nJ)	ACA Energy (nJ)	Total time (µs)
Input	2162689	219649	10%	31738	5615	6910
Guide	2359297	219650	10%	18482	6069	7539
Tmp0 (Hor0)	9437185	219650	2.3%	157170	88838	35248
Tmp1 (Ver0)	4818593	219650	4.5%	78584	22762	17624
Tmp2 (Hor1)	4718593	219650	4.6%	78586	22751	17724
Output (Ver1)	196609	1	0%	7553	456	1901
Total	**23692966**	**1098250**	**4.6%**	**372113**	**146491**	**86946**

When we scale up the memory sizes, the compression ratio increases. However, as can be seen, the ACA energy also increases. It is because we have implemented a fully flexible single cycle shift-registers. Therefore, the hardware complexity increases when we increase the size of the shift registers due to the intensive amount of wiring (any D-FF should connect to all the others). As it is mentioned before, it is possible to limit the shift amount in hardware or use multiple cycles to shift. This way the ACA circuit will be more scalable.

7.4.3. Software pipelining of guided filter application

In the previous execution method, we process each kernel sequentially. However, as it is clear from Figure 26, it is possible to execute them in parallel by exploiting a software pipelining

concept. As all the kernels execute similarly with the unique access pattern to the memory, in the parallel execution, we can use a longer word line in the memory to feed all the processes in parallel. The outcome is shown in Figure 27.

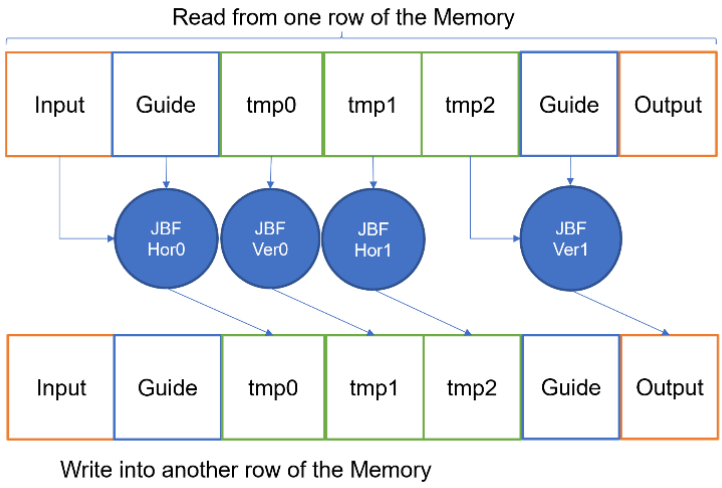

Figure 27 Parallel read/write from a wide memory in guided image filtering

In the software-pipelining format, one long word of the memory is read, processed, and write back to the memory. In this way, we save even more in address transactions. Processing one long word can take one or several cycles, dependent on the target compute architecture. In this case, ACA only needs to generate one address per line which results in a reduced cycle count and hence a higher performance and also energy efficiency for the address generation and the address and data communication network. However, the energy consumption for the memory access itself and the arithmetic instructions on the processor cores mainly remains the same. So the biggest gains are expected on the overall throughput and latency combined with a medium gain on the total energy consumption.

Table 6 results of guided image filtering when using software pipelining

Image size	The normal number of access	Compresses number of access (ACA)	Compression Ratio
32x32	368646	5958	1.6%
256x256	23692966	219654	0.92%

A more conventional computing architecture like a GPU can easily exploit software pipelining due to a high level of parallelism with SIMD (Single Instruction Multiple Data) structures. However, irregular memory access (same as most advanced image and video processing kernels), will cause inefficiency in SIMD processing and reduces the processor utilization. This problem can be solved by using an ACA like address decoding scheme.

Table 6 shows the compression ratio when using ACA. We will focus on the 256x256 image size to analyse this in more detail. Our parallel software pipelining based mapping can increase address instruction count by a factor of 5 when compared to the mapping from the more conventional loop kernel CUDA code which was discussed in Table 2 in subsection 4.2.

Note though that this heavily optimized software pipeline version is not feasible for GPU architectures as they are proposed today. Our colleagues from UGent have confirmed this. On top of this, even the compression of 22x for the CUDA code mapping in Table 2, is not directly reachable with any commercial GPU mapping, even for the most parallel GPU engines of today. Hence, we expect that compared to state-of-the-art GPUs, the guided filtering application can have at least 10x less address instructions. This compression will hence increase performance significantly. It also saves address instruction execution and address bus communication energy but we do not have a detailed model yet of the entire microarchitecture to allow us to accurately calculate that energy saving.

7.5. Conclusion

In conclusion, Nano-sim will make it feasible to run fast experiments with imec STT_MRAM technology. Even though the current release only supports memory read/write and ACA instructions, we can flexibly add further instructions to the CIM-A or CIM-P tile in the future. This platform can be useful for both internal use in imec and its partners, and for external use at the MNEMOSENE partners.

Evolving the new architectures by using compute in memory is a breakthrough in computer architecture. As most of the energy in STOA computer systems is consumed by data movement, compute in memory reduces total energy for a given performance target. It allows us to bring compute and memory close to each other and perform highly parallel computing. Besides, using NVM memory technologies reduces the power leakage of the system, especially when the SoC is memory dominant.

The main challenge in the CIM-A type compute-in-memory is the application level gain which is not yet sufficient compare to conventional technologies. One problem with using NVM is high energy consumption during write operations. This feature makes the technology infeasible for the write dominant applications. Many of the CIM-A instructions and architectures are built around NVM technologies. Therefore, for write-dominant applications, the in-memory process may not bring reasonable performance gain. However, processing in peripherals or near memory processing can be applied also to such memory technologies and then the context changes.

Another problem of CIM-A is the nature of analog signals. As the operations are done in the analog domain, it is prone to noise and variations. In this case, the application should be robust against these variations and it will be difficult to acquire standards for use in critical applications like health-care. This problem is more intense when using multi-level cells.

In future work, we should focus on the detailed micro-architecture level of CIMA and CIMP with realistic T/E models as well as the more application demonstrators. In this way, in imec, we are going to explore different microarchitecture-circuit-technology choices (STCO), including promising emerging memory options (especially MRAM and IGZO-DRAM) and global design PPAC trade-off exploration space.

8 Reflection and outlook beyond the project

In this report, our CIM tile architecture as well as the nano-instructions were presented. It should be noted that their definition is strongly influenced by the need to develop a simulator that allows for quick design space exploration between different memristor technologies investigated in the MNEMOSENE project. Furthermore, a simulator is more suited to be integrated with other simulators used/adapted/developed in other work packages. Our research performed and outlined in this deliverable is merely the starting point of more research and development in order to bring memristors to the market. Already at this moment, we are looking for future project (beyond MNEMOSENE) that will further improve and extend our work in MNEMOSENE. There are already several potential research directions and development tracks that we have identified and will pursue with the MNEMOSENE partners even after the end of this project. Examples are: multi-tile communications and prototyping of spiking neural networks using memristors.

References

[1] S. Yu and C. Pai-Yu, “Emerging memory technologies: Recent trends and prospects,” *IEEE Solid-State Circuits Magazine,* pp. 43-56, 2016.

[2] H. Jiang, C. Li, P. Lin, S. Pi, J. J. Yang and Q. Xia, “Scalable 3D Ta: SiOx Memristive Devices,” *Advanced Electronic Materials,* p. 1800958, 2019.

[3] A. Hardtdegen, C. La Torre, F. C{\"u}ppers, S. Menzel, R. Waser and S. Hoffmann-Eifert, “Improved switching stability and the effect of an internal series resistor in HfO 2/TiO x Bilayer ReRAM cells,” *IEEE Transactions on Electron Devices,* vol. 65, pp. 3229--3236, 2018.

[4] G. W. Burr, M. J. Brightsky, A. Sebastian, H.-Y. Cheng, J.-Y. Wu, S. Kim, N. E. Sosa, N. Papandreou, H.-L. Lung, H. Pozidis and others, “Recent progress in phase-change memory technology,” *IEEE Journal on Emerging and Selected Topics in Circuits and Systems,* vol. 6, pp. 146--162, 2016.

[5] S. Rashidi, M. Jalili and H. Sarbazi-Azad, “A survey on pcm lifetime enhancement schemes,” *ACM Computing Surveys (CSUR),* vol. 52, pp. 1--38, 2019.

[6] G. Snider, “Computing with hysteretic resistor crossbars,” *Applied Physics A,* vol. 80, pp. 1165--1172, 2005.

[7] J. Borghetti, G. S. Snider, P. J. Kuekes, J. J. Yang, D. R. Stewart and R. S. Williams, “‘Memristive’switches enable ‘stateful’logic operations via material implication,” *Nature,* vol. 464, pp. 873--876, 2010.

[8] H. Mahmoudi, T. Windbacher, V. Sverdlov and S. Selberherr, “Implication logic gates using spin-transfer-torque-operated magnetic tunnel junctions for intrinsic logic-in-memory,” *Solid-State Electronics,* vol. 84, pp. 191--197, 2013.

[9] S. Kvatinsky, D. Belousov, S. Liman, G. Satat, N. Wald, E. G. Friedman, A. Kolodny and U. C. Weiser, “MAGIC—Memristor-aided logic,” *IEEE Transactions on Circuits and Systems II: Express Briefs,* vol. 61, pp. 895--899, 2014.

[10] L. Xie, H. A. Du Nguyen, M. Taouil, S. Hamdioui and K. Bertels, “Fast boolean logic mapped on memristor crossbar,” in *2015 33rd IEEE International Conference on Computer Design (ICCD)*, IEEE, 2015, pp. 335--342.

[11] S. Li, C. Xu, Q. Zou, J. Zhao, Y. Lu and Y. Xie, “Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories,” in *Proceedings of the 53rd Annual Design Automation Conference*, 2016.

[12] M. Imani, Y. Kim and T. Rosing, “Mpim: Multi-purpose in-memory processing using configurable resistive memory,” in *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2017.

[13] C. Liu, B. Yan, C. Yang, L. Song, Z. Li, B. Liu, Y. Chen, H. Li, Q. Wu and H. Jiang, “A spiking neuromorphic design with resistive crossbar,” in *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2015.

[14] S. Hamdioui, L. Xie, H. A. Du Nguyen, M. Taouil, K. Bertels, H. Corporaal, H. Jiao, F. Catthoor, D. Wouters, L. Eike and others, “Memristor based computation-in-memory architecture for data-intensive applications,” in *2015 Design, Automation \& Test in Europe Conference \& Exhibition (DATE)*, 2015.

[15] T. Tang, L. Xia, B. Li, Y. Wang and H. Yang, “Binary convolutional neural network on RRAM,” in *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2017.

[16] P. Yao, H. Wu, B. Gao, S. B. Eryilmaz, X. Huang, W. Zhang, Q. Zhang, N. Deng, L. Shi, H.-S. P. Wong and others, “Face classification using electronic synapses,” *Nature communications,* vol. 8, pp. 1--8, 2017.

[17] Z. Wang, S. Joshi, S. Savel’ev, W. Song, R. Midya, Y. Li, M. Rao, P. Yan, S. Asapu, Y. Zhuo and others, “Fully memristive neural networks for pattern classification with unsupervised learning,” *Nature Electronics,* vol. 1, pp. 137--145, 2018.

[18] Z. Wang, C. Li, P. Lin, M. Rao, Y. Nie, W. Song, Q. Qiu, Y. Li, P. Yan, J. P. Strachan and others, “In situ training of feed-forward and recurrent convolutional memristor networks,” *Nature Machine Intelligence,* vol. 1, pp. 434--442, 2019.

[19] F. a. C. J. M. Cai, S. H. Lee, Y. Lim, V. Bothra, Z. Zhang, M. P. Flynn and W. D. Lu, “A fully integrated reprogrammable memristor--CMOS system for efficient multiply--accumulate operations,” *Nature Electronics,* vol. 2, pp. 290--299, 2019.

[20] A. Siemon, S. Ferch, A. Heittmann, R. Waser, D. Wouters and S. Menzel, “Analyses of a 1-layer neuromorphic network using memristive devices with non-continuous resistance levels,” *APL Materials,* vol. 7, p. 091110, 2019.

[21] L. Xie, H. A. Du Nguyen, J. Yu, A. Kaichouhi, M. Taouil, M. AlFailakawi and S. Hamdioui, “Scouting logic: A novel memristor-based logic design for resistive computing,” in *2017 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2017.

[22] A. a. D. R. Siemon, M. Schultis, X. a. L. E. Hu, A. Heittmann, R. Waser, D. Querlioz, S. Menzel and J. Friedman, “Stateful Three-Input Logic with Memristive Switches,” *Scientific reports,* vol. 9, pp. 1--13, 2019.

[23] A. Siemon, D. Wouters, S. Hamdioui and S. Menzel, “Memristive Device Modeling and Circuit Design Exploration for Computation-in-Memory,” in *2019 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2019.

[24] A. Siemon, S. Menzel, D. Bhattacharjee, R. Waser, A. Chattopadhyay and E. Linn, “Sklansky tree adder realization in 1S1R resistive switching memory architecture,” *The European Physical Journal Special Topics,* vol. 228, pp. 2269--2285, 2019.

[25] A. Sebastian, M. Le Gallo, R. Khaddam-Aljameh and E. Eleftheriou, “Memory devices and applications for in-memory computing,” *Nature Nanotechnology,* pp. 1--16, 2020.

[26] H. A. D. Nguyen, J. Yu, M. A. Lebdeh, M. Taouil, S. Hamdioui and F. Catthoor, “A classification of memory-centric computing,” *ACM Journal on Emerging Technologies in Computing Systems (JETC),* vol. 16, pp. 1--26, 2020.

[27] K. a. A. N. a. H.-E. S. Fleck, V. Longo, F. Roozeboom, W. Kessels, U. B{\"o}ttger, R. Waser and S. Menzel, “The influence of non-stoichiometry on the switching kinetics of strontium-titanate ReRAM devices,” *Journal of Applied Physics,* vol. 244502, p. 120, 2016.

[28] K. Fleck, U. B{\"o}ttger, R. Waser, N. Aslam, S. Hoffmann-Eifert and S. Menzel, “Energy dissipation during pulsed switching of strontium-titanate based resistive switching memory devices,” in *2016 46th European Solid-State Device Research Conference (ESSDERC)*, 2016.

[29] 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, pp. 4304--4312, 2018.

[30] S. Nandakumar, M. L. Gallo, C. Piveteau, V. Joshi, G. Mariani, I. Boybat, G. Karunaratne, R. Khaddam-Aljameh, U. Egger, A. Petropoulos and others, “Mixed-precision deep learning based on computational memory,” *arXiv preprint arXiv:2001.11773,* 2020.

[31] W. Gallagher, E. Chien, T.-W. Chiang, J.-C. Huang, M.-C. Shih, C. Wang, C.-H. Weng, S. Chen, C. Bair, G. Lee and others, “22nm STT-MRAM for Reflow and Automotive Uses with High Yield, Reliability, and Magnetic Immunity and with Performance and Shielding Options,” in *2019 IEEE International Electron Devices Meeting (IEDM)*, 2019.

[32] L. Wu, S. Rao, G. C. Medeiros, M. Taouil, E. J. Marinissen, F. Yasin, S. Couet, S. Hamdioui and G. S. Kar, “Pinhole defect characterization and fault modeling for STT-MRAM testing,” in *2019 IEEE European Test Symposium (ETS)*, 2019.

[33] L. Kull, T. Toifl, M. Schmatz, P. A. a. M. C. Francese, M. Braendli, M. Kossel, T. Morf, T. M. Andersen and Y. Leblebici, “A 3.1 mW 8b 1.2 GS/s single-channel asynchronous SAR ADC with alternate comparators for enhanced speed in 32 nm digital SOI CMOS,” *IEEE Journal of Solid-State Circuits,* vol. 48, pp. 3049--3058, 2013.

[34] I. L. M. R. K. a. L. B. A. Rahimi, “A fully-synthesizable single-cycle interconnection network for Shared-L1 processor clusters,” in *Design, Automation & Test in Europe*, 2011.

[35] C. L. P. L. S. P. J. J. Y. a. Q. X. H. Jiang, “Scalable 3D Ta: SiOx Memristive Devices,” *Advanced Electronic Materials,* 2019.

[36] J. S. a. X. T. K. He, “Guided Image Filtering,” *IEEE Transactions on Pattern Analysis & Machine Intelligence,* 2013.

[37] I. L. M. R. K. a. L. B. A. Rahimi, “A fully-synthesizable single-cycle interconnection network for shared-L1 processor clusters,” in *Design, Automation and Test in Europe*, 2011.