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: 11/10/2017

D3.4 – Report on Mapping of Micro-Kernels to Macro-Architecture

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:	ETH Zurich (ETHZ)
Report Issue Date:	30/06/2020

Document History *(Revisions – Amendments)*	
Version and date	**Changes**
1.0 30/06/2020	First version

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

1. Introduction

In this deliverable, we provide an overview and explanation of the process of mapping micro-kernels as part of applications as defined in D1.1 to operations to be performed by compute-in-memory (CIM) accelerators. Additionally, we provide an evaluation of several performance metrics of the mapped micro-kernels and compare it to a software-only baseline with different implementation with different degrees of optimization when executed on a general-purpose RISC-V core. The document is structured into seven sections. In Section 1 and 2 we summarize the system's architecture and the programming model. In section 3 the available simulation environments for performance evaluation is introduced. In sections 4 and 5 we introduce the example kernel and methodology for the subsequent analysis and conclusion in sections 6 and 7.

2. System Overview

2.1 Terminology

In this subsection, we reintroduce the terminology used throughout the rest of this document and past deliverables part of the MNEMOSENE project.

Nano Architecture/Nano Instruction At the lowest abstraction layer we have the Nano Architecture. It describes the architecture used to interact with the non-volatile memory (NVM) crossbar. The Nano Instructions encode the individual steps in the procedure to interact (i.e. access) the cells in a single NVM crossbar at the level of individual control signals.

Micro Architecture/Micro Instruction The Micro Architecture wraps the Nano Architecture and thus still contains a single NVM crossbar. It contains the necessary digital peripherals like buffers, state machines, and multiplexers to perform micro instructions; a more high-level interaction with the NVM crossbar (i.e. perform CIM operation *X* on rows *k* to *j*). It also translates externally supplied Micro Instructions to a sequence of nano instructions to control the individual bitlines of the crossbar and activate the analog/mixed-signal peripherals in the right order to perform the desired operation.

Macro Architecture At the highest level the macro architecture describes how the micro architecture is embedded into a larger system combining memory, conventional compute resources (e.g. processor cores) with one or multiple NVM crossbars for CIM operation.

CIM Tile With the term CIM-Tile we refer to a single instantiation of the Micro Architecture *without* the necessary digital glue logic to integrate the instance into the Macro Architecture.

CIM Accelerator A CIM-Accelerator is the combination of a CIM Tile with interface glue logic for instantiation in the Macro Architecture. The most notable component part of this glue logic would be means of transferring data to and from the COM Tile, for example a direct memory access (DMA) module that interfaces the global shared memory of the Macro Architecture and generates streams of data in the format expected by the CIM Tile.

2.2 Macro Architecture

In this subsection, we give a quick overview of the Macro Architecture. For a more in-depth explanation of the internals of the Macro Architecture we refer to deliverable D3.2.

Figure 1 depicts the macro architecture of a system containing several conventional processor cores and multiple CIM Accelerators. The system is split into two different power/clock

domains: The Cluster domain contains several general-purpose processing cores, CIM Accelerators, and shared memory. The SoC domain on the other hand contains IO blocks for communication with off-chip peripherals and external memory, a power management unit, and clock generators. All these components as well as the cluster domain is controlled by a tiny processor, the so-called fabric controller.

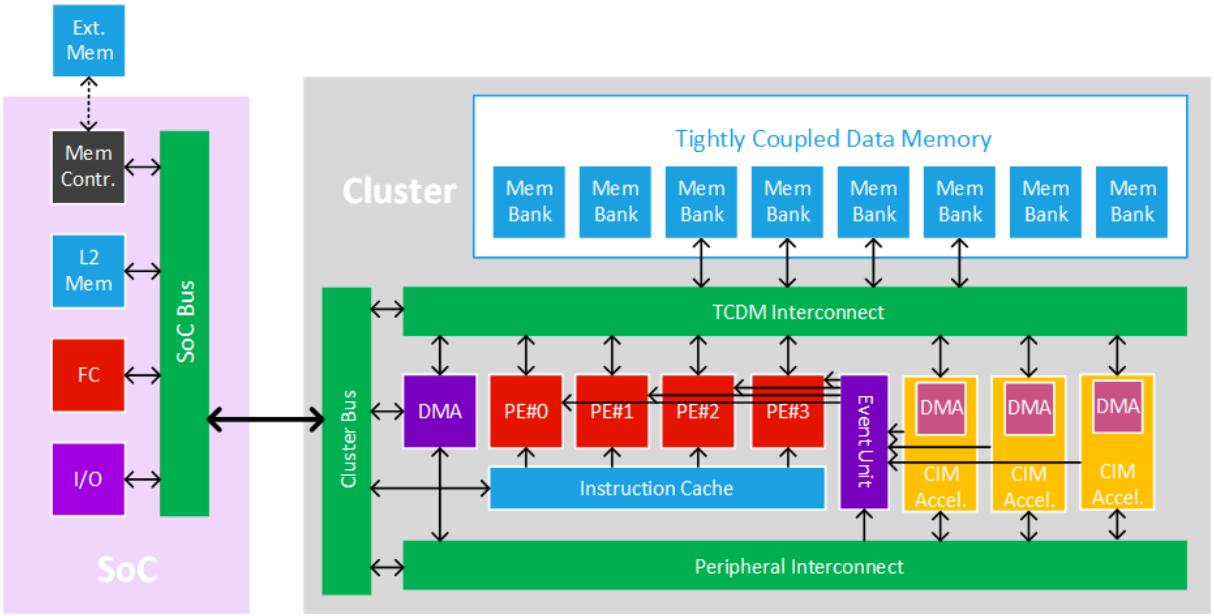

Figure 1 Block Schematic overview of Macro Architecture based on the OpenPULP platform

This organization is derived from the open source OpenPULP, which was developed by ETH Zurich as part of the PULP Platform and is used as a representative digital system that interfaces with CIM accelerators. Throughout this deliverable we will use conventional (i.e. without COM accelerator) implementations of the same architecture as a baseline to demonstrate the working order, feasibility and highlight the differences between In-Memory computing and traditional computing.

2.3 Fabric Controller

The fabric controller, in previous PULP architectures based on a very lightweight RISC-V core (zeroRISCY), is responsible for orchestrating I/O peripherals the external memory controller as well as the cluster. To process data on the high-performance cluster the fabric controller can fork execution to the general-purpose cores within the cluster and can dynamically adjust the frequency of the cluster for optimal performance and energy efficiency.

2.4 Cluster Architecture

The cluster consists of several general-purpose Processing Elements (PE) based on ETH's RI5CY RV32IMF[1] core and one or multiple CIM Accelerators. All processing elements share access to the tightly-coupled data memory, a multi-banked software managed scratchpad memory for data exchange between the processing elements[2]. The DMA module within the cluster can be programmed by the Processing Elements to transfer data between the larger

[1] As of June 2020, this core is being actively maintained by OpenHW group under the name CV32E40P.
[2] This architecture has been shown to achieve SoA results in [6].

L2 memory in the SoC domain and an event unit provides the means for synchronization between the processing elements. Each processing element is connected via two interconnects: The peripheral interconnect is a low bandwidth bus based on AMBA APB1 used for configuration of the DMA, event unit, and most importantly the CIM Accelerators. The TCDM interconnect provides low latency and high-bandwidth access to the shared TCDM memory and is based on the so-called Logarithmic Interconnect, a forest of arbitration trees providing parallel single-cycle access to the multi-banked memory with transparent arbitration in the case of bank conflicts between different processing elements.

2.5 CIM Accelerator

The CIM Accelerator contains a CIM Tile and interface-logic that provides arbitration and synchronization of requests from multiple cores to the same CIM Accelerator and hides the tasks of address generation and data realignment from the CIM Tile.

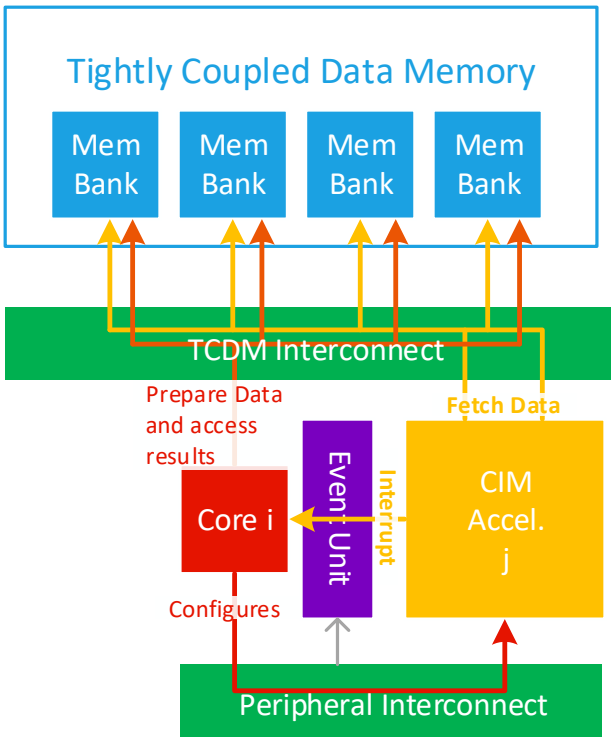

Figure 2 Dataflow between conventional processing core and CIM Accelerator

Figure 2 illustrates how one of the general-purpose cores, *Core i* can offload a micro instruction to one of the CIM Accelerators, in this example *CIM Accelerator j*. Software running on *Core i* pre-processes and prepares the data that is to be processed by the *CIM Accelerator j* by writing it to the shared memory. Then the core writes the micro-ISA instruction and its corresponding arguments (e.g. addressing scheme, matrix dimensions etc.) to memory-mapped configuration registers within the *CIM Accelerator j* via the *Peripheral Interconnec*t. From the core's point of view this is performed using conventional load/store instructions with appropriately chosen addresses that are assigned to the memory-mapped *CIM Accelerator j*. After the core is done configuring the accelerator it triggers the execution of the micro instruction again through writing to a dedicated register of the *CIM Accelerator j*. The *CIM Accelerator j* can then independently fetch the input data (e.g. weights to write to the non-

volatile memory or vector elements to process) from the shared memory and write back the results to the shared memory according to the memory access scheme configured by the core. Once the *CIM Accelerator j* finishes the execution of the micro instruction it sends an event signal to the event unit which in turn interrupts *Core i* that offloaded the instruction. Finally, the core can then post-process the results, start another micro instruction or issue a DMA transfer to move them out of the cluster into global memory. The architecture allows for an arbitrary number of Cores (*i*) and CIM Accelerators (*j*) to co-exist and work in concert. The exact number of *i* and *j* can be determined on the workload expected as well as individual capabilities of the CIM Accelerators. For most operations, it is feasible that a single core controls and configures multiple CIM Accelerators (i.e. one core per 4 to 8 CIM Accelerators), whereas for problems that contain operations that do not translate well to in memory computing, additional Cores can be used to post/pre-process the data stored in the global memory in parallel.

The CIM Tile supports various operations, e.g. vector-matrix multiplication (VMM), bulk Boolean bitwise (BBB) operation. It can do so thanks to a finite state machine that controls the both analog as well as digital components inside a tile. For the CIM tile to perform any operation, e.g. writing, reading, VMM, BBB, first a core should program the configuration register. It is done through some simple load-store instructions as every peripheral is memory mapped. Then, the FSM takes the command and directs the operation, specified in the configuration, based on the parameters that are set in the configuration register. These operations have been discussed in detail in Deliverable 3.3.

3. Programming Model

The CIM Accelerator exposes the high-level CIM functions- read/write to CIM array and vector-matrix multiplication to the system via register-interface (i.e. configuration and control registers as memory-mapped regions). Mapping of high-level application (written in C/C++) to CIM accelerator at system levels involves three steps.

1. Preparing data on shared data memory (scratchpad/main memory) where CIM accelerator can read from,

2. Configuring the CIM Accelerator for the desired operation through configuration and control registers

3. Waiting for the completion of CIM computation and reading back the results.

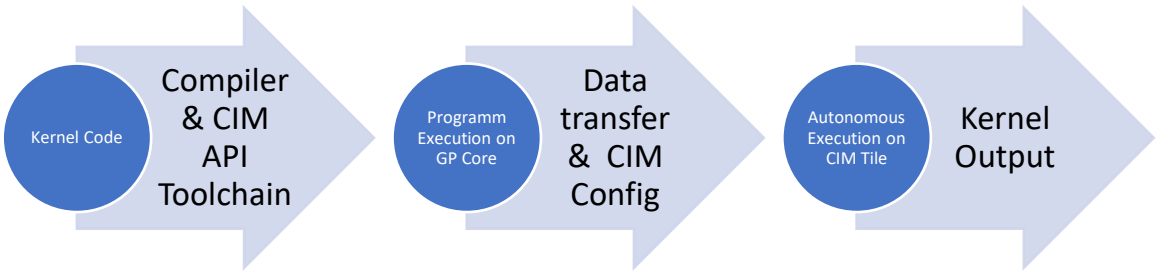

Figure 3 Kernel Mapping and execution flow

Preparing data and reading back the result to/from CIM Accelerator is simply a process of copying data from global or private memory (CPU cache) to a CIM Accessible memory and vice-versa. The software managed TDCM scratchpad is considered as an ideal place for sharing data to CIM due to its property, a low-latency and high-bandwidth memory hierarchy. Existing PULP software libraries can be reused for allocating and managing these scratchpad memories efficiently. The CIM accelerator exposes the configuration and control register as a memory-mapped interface. Therefore, in order to offload computation to CIM, the host needs to write proper configurations to this memory-mapped region. As part of MNEMOSENE a dedicated software library has been developed to simplify this offloading process. As shown in Figure 4, the library exposes high-level API for CIM micro-kernels to the programmer and hides the low-level platform-specific communication model. Once the CIM receives all the necessary parameters, it starts its execution and runs in parallel to the host processor. The status of the CIM can be monitored with the status registers on the host processor, and the result can be read back once the CIM completes the offloaded micro-kernel. The overall flow of the micro-kernel offloading is shown in Figure 3.

```
/* BLAS like library for CIM offloading
 * Terminology:
 *    Gemm: C[M][N] = A [M][K] x B[K][N]
 *    lda, ldb, ldc: Leading dimension of A, B, C tensors (refer BLAS)
 */
// 1. Store A[M][K] to XBar(x_pos, y_pos)
void store_gemm(uint8_t **A, uint32_t M, uint32_t K, uint32_t lda,
                uint32_t x_pos, uint32_t y_pos);

// 2. Read B[K][N] from XBar(x_pos, y_pos)
void read_gemm(uint8_t **B, uint32_t K, uint32_t N, uint32_t ldb,
               uint32_t x_pos, uint32_t y_pos);

// 3. MMM: C[M][N] =  A[M][K] * XBar[K][N] @(x_pos, y_pos)
void cim_gemm(uint8_t **A, uint8_t **C,
              uint32_t M, uint32_t N, uint32_t K,
              uint32_t lda, uint32_t ldc,
              uint32_t x_pos, uint32_t y_pos);
```

Figure 4 Overview of CIM Library

Host-only

```
void main() {
  // Matrix multiplication
  for (int m=0; m<M; m++) {
    for (int n=0; n<N; n++) {
      for (int k=0; k<K; k++) {
        C[m][n] += A[m][k] * B[k][n];
      }
    }
  }
}
```

Host + CIM

```
void main() {
  // Prepare data on scratchpad (with pulp libs)
  A_shared = cim_memcpy(M * K, A);
  B_shared = cim_memcpy(K * N, B);
  C_shared = cim_malloc(M * N, C);

  // Matrix multiplication
  store_gemm(A_shared, M, K, 0 /*lda*/,
                    0 /*x_pos*/, 0 /*y_pos*/);
  cim_gemm(A_shared, C_shared, M, N, K,
        0 /*lda*/, 0 /*ldc*/, 0 /*x_pos*/, 0 /*y_pos*/);
  cim_wait_done();
}
```

Figure 5 Overview of CIM Offloading

4. Simulation Environment

One of the key challenges of MNEMOSENE is simulating a system that currently is not physically manufactured and furthermore as a new technological concept is not yet integrated in common design flows.

As part of MNEMOSENE we have adapted two parallel approaches for simulation and/or emulation environments capable of delivering performance metrics on micro-kernel execution within the scope of the Macro Architecture.

4.1 RTL Model

For the Macro Architecture, excluding the CIM Tile itself, there exists a register-transfer-level model implemented in SystemVerilog. Being synthesizable and silicon-proven the model provides cycle-accurate numbers for execution time and allows fairly precise area estimates of the Macro Architecture's digital logic for a given target technology node. While it gives the utmost accuracy in execution time, simulation using the RTL model is time consuming and is not well suited to executing larger kernels with several millions of cycles to simulate. This is why a second Cycle Approximate Model has been developed.

4.2 Cycle Approximate Model

In order to speed-up simulations in RTL model, the cycle approximate model has been developed that exploits a trade-off between simulation speed and accuracy in the number of cycles. The Cycle Approximate Model is a loosely-timed C++ implementation of the Macro Architecture with an interface for SystemC co-simulation. The speed up comes mainly from the fact that the model only approximates the effects of memory contention in the multi-banked shared memory. As the model still accurately models stalls and hazards of the instructions in the general-purpose cores and a multi-banked memory architecture with interleaved mapping is used, cycle count errors are less than 10%, while the simulation execution speed increases at least an order of magnitude. Most importantly, as a C++ implementation it can run independently from EDA tools and as an open source model can be adapted to interface with other simulators such as detailed technology models and simulators being developed as part

of MNEMOSENE. At time of writing the integration of the CIM Tile's behavioural SystemC model is nearing its final stages of completion.

5. Methodology

In the following subsections, we will specify the evaluated kernel for CIM mapping performance analysis, define the performance metrics we considered, introduce the baseline implementation used for comparison and highlight the methodology followed to obtain the performance figures.

5.1 Evaluated Kernel

For this deliverable we use the general matrix multiply (gemm) kernel, which is at the heart of many signal processing and deep-neural-network (DNN) algorithms, to have a more in-depth analysis of the energy-efficiency advantages of the CIM approach. Given two input matrices, A and B of dimension l x m and m x n the kernel computes the matrix product C of dimension l x n. The kernel has time complexity O(lmn) and in it's most naive implementation can be expressed with the following lines of C code:

```
1. void matmul(int *A, int *B, int *C)
2. {
3.   for (int i = 0; i < L; i++) {
4.     for (int j = 0; j < M; j++) {
5.       C[i*M+j] = 0;
6.       for (int k = 0; k < N; k++) {
7.          C[i*M+j] += A[i*N+k] * B[k*M+j];
8.       }
9.     }
10.   }
11. }
```

Listing 1 GEMM Kernel implementation in C

While there exist many alternative implementations (e.g. by decomposing the operand matrices into block matrices) those approaches mainly target to improve performance by exploiting data-locality. However, in a system such as the OpenPULP, which does not rely on caches but works with low-latency shared memory, data locality has no impact on performance and may even result in execution overhead.

5.2 Performance Metrics

Throughout the rest of this document we consider the following performance metrics:

- Operations/Second
- Energy/Operation
- Area

For the operation count, we only consider the useful computation in the body of the innermost loop of the GEMM kernel counting additions and multiplications individually. Address calculation operations are not considered as they would only apply for the software implementation. With this definition, the GEMM kernel has 2 operations per innermost loop iteration and thus needs 2 x l x m x n operations to complete.

5.3 Baseline

As a baseline for comparison with the CIM Tile mapped performance of the gemm kernel, we consider the metrics achieved on one of the Macro Architecture's general-purpose cores. The cluster domain within the macro architecture contains up to 8 CV32E40P cores (formerly known as RI5CY). CV32E40P is an energy-efficient 32-bit in-order core with a 4-stage single-issue pipeline and latch memory-based register-file for the 32+32 (base+floating point extension) registers. It supports the RISC-V RV32I base ISA with the following standard ISA extensions:

Extension	Description
C	Support for Compressed (16-bit) Instructions
M	Support for Integer Multiplication and Division
Zicsr	Control and Status Register Instructions
Zifencei	Instruction-Fetch Fence for JIT compiled code
F	Single-precision floating point Instructions

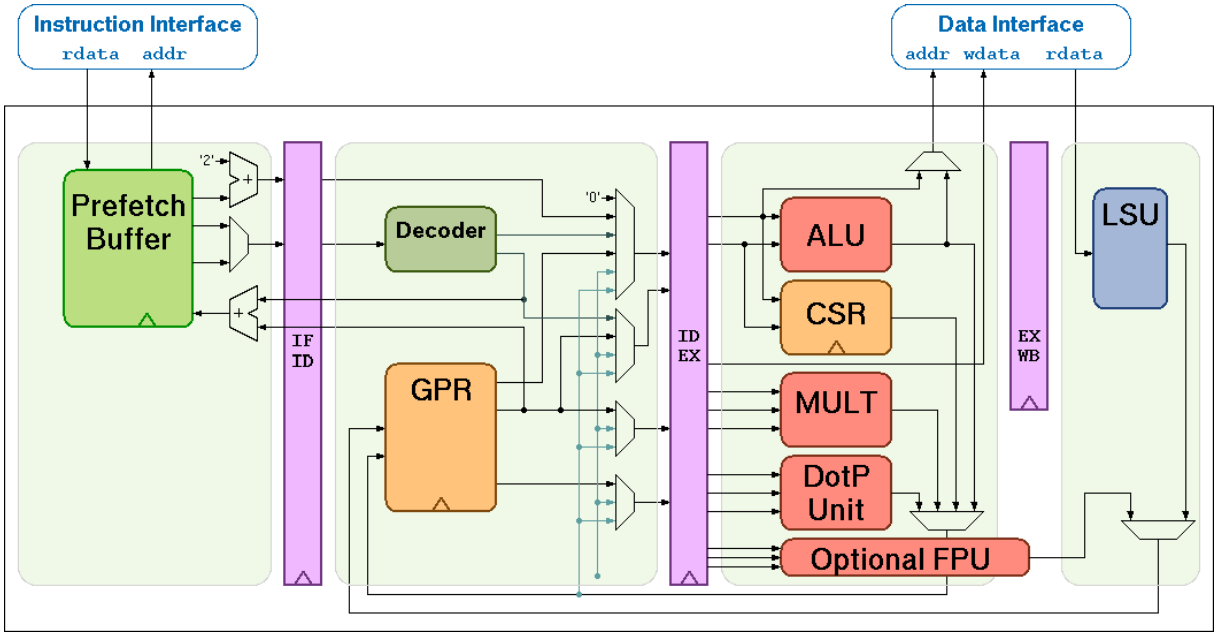

Figure 6 Pipeline structure of the CV32E40P core used as a baseline for kernel performance comparison

Additionally, the core supports a number of non-standard ISA instructions as part of the RISC-V custom extension Xpulphwlp. In the following paragraph, we limit ourselves to the introduction of only those additional instructions which have an effect on gemm kernel execution performance;

Post-increment Load/Store The post-increment load/store instructions perform a load, or a store, respectively, while at the same time incrementing the address that was used for the memory access. Since it is a post-incrementing scheme, the base address is used for the access and the modified address is written back to the register-file. These instructions improve performance especially in kernels with a linear memory access scheme since no additional cycle is needed for pointer updating in the innermost loop.

Hardware Loop Instructions CV32E40P supports hardware loops also known as zero-overhead loops. The corresponding instructions cause the core to enter a loop with fixed iteration count (provided as a register argument) without the overhead of branch condition checking and loop counter update. Up to two such loops can be nested. This feature has a positive effect on every kernel that uses thigh nested loops with a data-independent iteration count.

SIMD Instructions In cases where 16- or 8-bit precision is sufficient, an additional speedup of the software implementation can be achieved by using single-instruction-multiple-data (SIMD) extensions to the base integer instruction set. They allow to perform 2x 16-bit or 4x 8-bit ALU operations in a single cycle by packing multiple operands in a single 32-bit register. Most relevant for the baseline 8-bit gemm kernel execution is the support of the 8-bit sum dot product instruction that performs 4 multiply-accumulate (MAC) operations in a single cycle.

To acquire the cycle count to calculate the throughput and energy-efficiency numbers we execute the GEMM kernel on the Cycle Approximate Model and use the core's hardware performance counters to track the number of cycles spent.

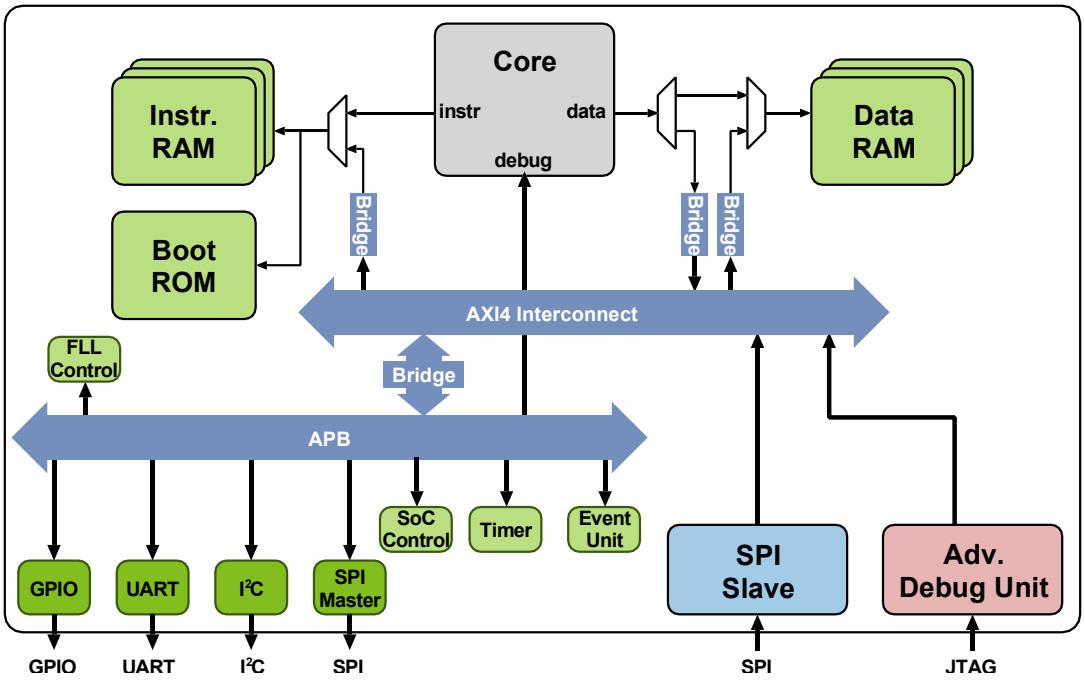

Figure 7 Overview of the baseline architecture

Power and Area Analysis have been extracted from previous work on the CV32E40P core where the core was integrated within an instances of the open-source PULPino micro-controller platform [1], [2]. The micro-controller has one core, one data and one instruction memory, an event unit for handling interrupts and common peripherals as SPI, UART, I2C, GPIO and timers. In Figure 7 we show an overview of the PULPino architecture used for evaluation. The PULPino instances have been fully synthesized, placed and routed in UMC 65nm at the 1.08V worst-case corner using Synopsys Design Compiler v2015.06 (DC) for the synthesis and Cadence Innovus v15.20 (INNOVUS) for the place and route phase.

Using the average power figures from post-layout power simulation for the 2D-Convolution workload which has a very similar activity signature to the gemm kernel we estimate the energy/operation of the evaluated kernel.

5.4 CIM Accelerator

For the crossbar within the CIM tile we consider numbers reported by IBM for their 4-bit PCM crossbar[3]. To mimic an 8-bit weight with 4-bit cells, two columns are used, one for four MSBs and the other for four LSBs. The result is computed by a weighted sum of MSB and LSB columns in the digital logic within the Micro Architecture. To model the energy consumption of the mixed-signal periphery logic (ADC, DAC, S+H etc.) we use the reported numbers from Ali Shaffiee et.al [4]. For more details about the CIM Accelerator internals please refer to D3.3 and D3.1.

Since the integration of the CIM Accelerator model into the Cycle Approximate Model of the Macro Architecture is still work in progress, a two-step approach was followed to obtain the cycle count of the CIM Accelerator; The time and energy spent on the actual crossbar write operation and readout (thus vector-matrix multiplication) can be calculated using the numbers reported by IBM[3]. The execution time overhead of for CIM Accelerator configuration and data transfer is obtained by multiplying the number of necessary configuration transactions from the conventional processing core to the CIM Accelerator with the number of cycles to complete one such transaction. The concrete number of cycles to perform these transactions is obtained from simulation on the RTL model of the Macro Architecture.

The energy and area numbers for the CIM Accelerator's digital interface logic are obtained after synthesis with Synopsys Design Compiler, version 2016.12. A 28 nm process is used at 0.95 V operating voltage and 25°C temperature process corner. For power consumption analysis, switching activity information files (SAIFs) are generated with ModelSim 10.5 are obtained after synthesis with Synopsys Design Compiler, version 2016.12. A 28 nm process is used at 0.95 V operating voltage and 25°C temperature process corner. For power consumption analysis, switching activity information files (SAIFs) are generated with ModelSim 10.5.

6. Results

6.1 Baseline Performance

Processor Core	*CV32E40P without extension*	*CV32E40P with extension*
Vdd [V]	1.08	1.08
Max. Freq [MHz]	357	357
Avg. Power [mW]	9.38	10.38
Area [kGE]	46.9	53.5
Area [µm^2]	68'000	77'000

Table 1 Frequency, Area and Power Consumption of CV32E40P in 65nm with and without ISA extensions

Table 1 shows the power and area figures of the core with and without the custom ISA extensions in UMC 65nm [5]. The extended architecture increases in area by 6.6 kGE and increases the average power consumption by 14%.

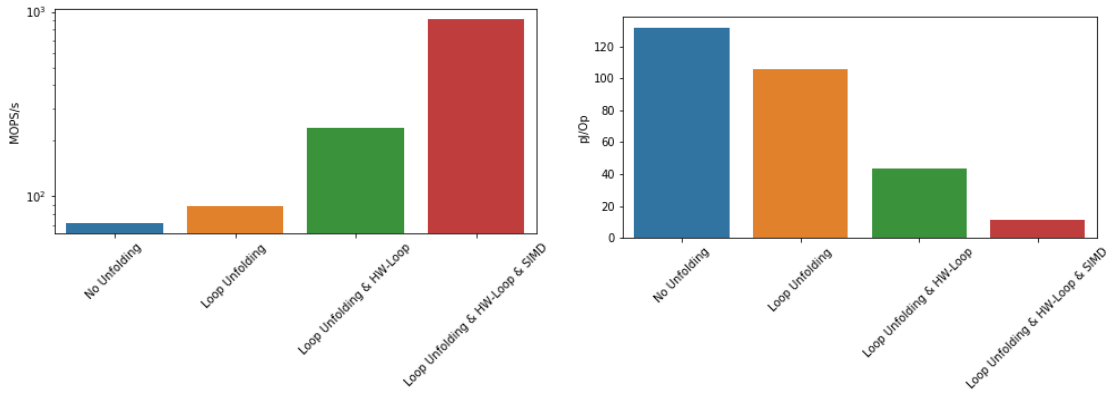

Figure 8 Throughput and energy efficiency of the gemm kernel on CV32E40P in UMC65nm at 1.08V, 357MHz

Figure 8**Error! Reference source not found.** shows the cycle count of the gemm kernel for 8-bit square matrices of size 256 by 256 and different degrees of optimization. The “No Unfolding” case is the result of executing the code in Listing 1 with optimization level 3 (gcc -o3). Some efficiency gain can be achieved by reducing the amount of data hazards in the innermost loops by means of manual loop unfolding. This technique already increases throughput and energy efficiency by 24%.

Even higher gains can be achieved by enabling automatic usage of the hardware loop and post-increment load instructions. Listing 2 shows the assembly instructions of innermost loop of the gemm kernel with and without ISA extensions.

```
...
1c0412c6:  lw      a4,0(a5)
1c0412c8:  lw      a1,0(a3)
1c0412ca:  addi    a5,a5,4
1c0412cc:  addi    a3,a3,1024
1c0412d0:  mul     a4,a4,a1
1c0412d4:  add     a2,a2,a4
1c0412d6:  bne     a0,a5,1c0412c6
...
```

```
...
00c8c0fb  lp.setup    x1,a7,1c04138e
1c04137a: p.lw        a2,8(t1!)
1c04137e: p.lw        a4,8(t3!)
1c041382: p.lw        a3,s8(t5!)
1c041386: p.lw        a5,s8(t4!)
1c04138a: pv.sdotsp.b a6,a2,a3
1c04138e: pv.sdotsp.b a6,a4,a5
...
```

Listing 2 Assembly listing of the innermost loop of the gemm kernel with (a) and without (b) custom ISA extension and 2-fold loop unrolling

As mentioned above, support for these custom instructions comes with an increase in area and power, but the efficiency gains far outweigh these costs; The energy per operation decreases by 59% with the hardware loop and post-increment load instructions while usage of the 8-bit SIMD dot product instruction achieves another efficiency boost of 3.87x. The most energy efficient operating point of our baseline architecture is thus at **11.26pJ/Op** with a throughput of **909 MOPS/s**.

These numbers represent a fairly optimized implementation for a single core processing system using several optimization steps and is close to SoA.

6.2 CIM Tile Performance

Since the operation of the programming memristor crossbar takes a long time, 2.5 µs according to an IBM prototype, it is only reasonable to perform a computation, e.g. VMM, using the memristor crossbar only if the number of input vectors are big enough. Here we assume a matrix-matrix multiplication on an 8-bit input matrix of size 256x256, and an 8-bit weight matrix of size 256x256. Note that, this represents a fairly conservative tile size, our initial estimations show that much larger CIM tiles (i.e. 4096 x 256) would be feasible which will further increase the gains of in memory acceleration.

Crossbar Parameter	**Value**	
Memristor Technology	IBM PCM (90nm)	
Cell precision	8-bit (implemented by 2×(4-bit) PCMs)	
Compute and Write Latency/8-bit	1 µs and 2.5 µs	
Compute Energy/8-bit	200 fJ (2x100 fJ/4-bit PCM)	
Write Energy/8-bit	200 pJ (2x100 pJ/4-bit PCM)	
Area (256×256)	200 μm^2	
Peripheral Circuitry (for 1us readout period)	**Energy**	**Area**
Mixed Signal Circuitry	2.1 nJ/cycle (@1.2GHz)	1252 μm^2
Micro-Architecture (Digital Peripherie)	64.8 pJ/byte	865 μm^2
Total	**Value**	
Compute (Read) Power	16.2 mW	
Write Power	33.5 mW	
Area	2317 μm^2	

Table 2 CIM Tile Area and Energy figures

Figure 9 shows the overall execution time of the 8-bit 256 x 256 gemm kernel considering a Macro Architecture clock frequency of 357 MHz identical to the baseline architecture; To prepare the CIM Accelerator for gemm kernel execution we need 11 write transactions to the configuration registers. RTL simulation of the platform showed that the core needs on average about 4 cycles to perform a single write transaction to the configuration interface of the CIM Accelerator. Fetching the weight matrix from shared memory is performed by the DMA over the low-latency interconnect. Since this requires only a single cycle per 32-bit read and write (4x 8-bit data) a total of 16’384 cycles is spent for weight data fetching. Programming the weight to the crossbar requires 2.5 µs per row which translates to 228’480 cycles in the macro architectures base clock. Performing the actual gemm computation is then performed by repeated vector matrix multiplication which accounts for another 16’384 cycles of data fetching and 1µs for the crossbar read operation and thus a total of 107’776 cycles.

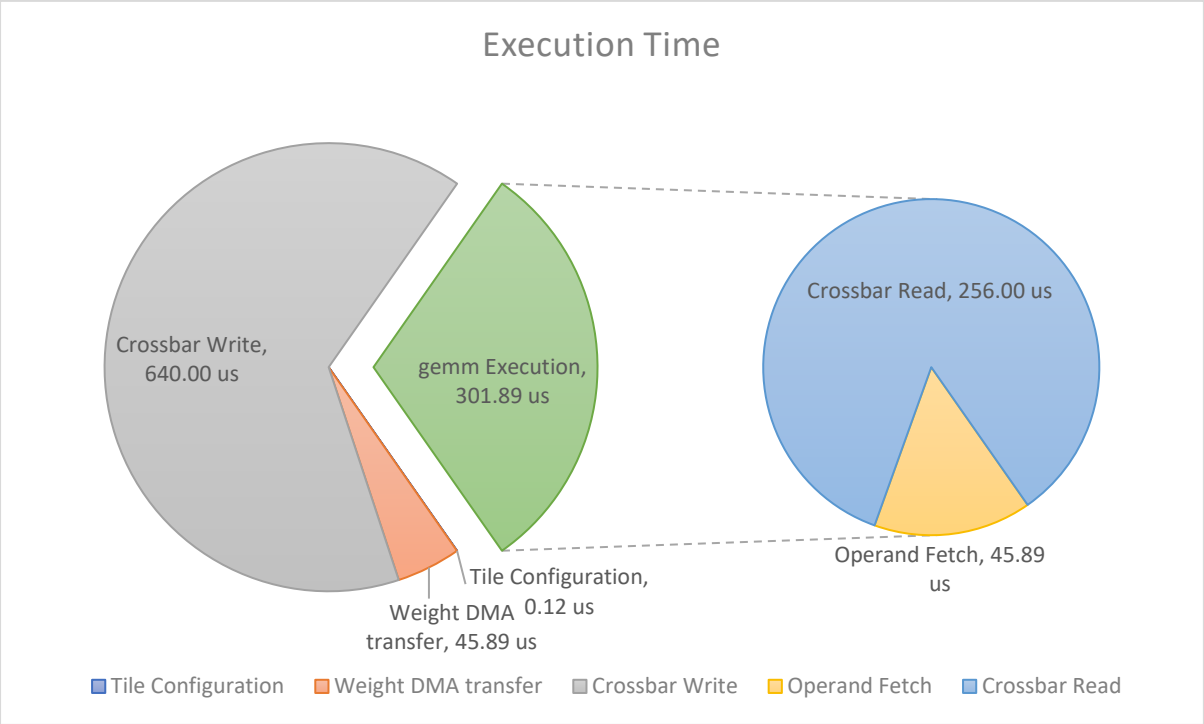

Figure 9 Breakdown of gemm Execution time of CIM Accelerator

Table 3 provides the overall estimated energy efficiency and throughput of the gemm kernel execution on the CIM Accelerator. The CIM approach provides a 6.78x improvement in terms of energy efficiency compared to the baseline implementation in the scenario where the second matrix of the gemm kernel needs to be changed in each invocation. An additional 5.7x improvement can be achieved in applications where one of the matrices is static thus hiding the high setup costs of programming the crossbar. For these scenarios, an overall efficiency improvement of up to 38.7x compared to the baseline implementation can be achieved. We note that this efficiency is comparing an advanced baseline to a modest CIM implementation, and therefore should be considered closer to the lower bound of realistic implementations.

	Gemm Execution (including x-bar write)	**Gemm Execution (excluding x-bar write)**
Throughput	16.98 GOPS/s	55.57 GOPS/s
Energy Efficiency	1660 fJ/Op	291 fJ/Op

Table 3 Throughput and Energy Efficiency of CIM Tile Execution

7. Conclusions

This deliverable explains the process of mapping common kernels to realistic digital systems enhanced by in memory computing accelerators and shows the result of our first experiments in mapping micro-kernels to such an architecture as part of the MNEMOSENE project taking into account the overheads involved in communicating to and from the CIM accelerators and reporting on implementations for which more automated steps are underway. Our results clearly show that when compared to SoA digital only implementations, order of magnitude improvements are indeed attainable even for smaller systems that do not fully exploit the advantages of in memory computing systems. Specifically, we see under realistic conditions a 38.7x improvement in energy efficiency.

8. References

[1] Andreas Traber, Michael Gautchi, „PULPino: Datasheet“. Juni 09, 2017, Zugegriffen: Juni 25, 2020. [Online]. Verfügbar unter: https://pulp-platform.org/docs/pulpino_datasheet.pdf.

[2] P. Davide Schiavone *u. a.*, „Slow and steady wins the race? A comparison of ultra-low-power RISC-V cores for Internet-of-Things applications“, in *2017 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, Sep. 2017, S. 1–8, doi: 10.1109/PATMOS.2017.8106976.

[3] M. L. Gallo *u. a.*, „Compressed sensing with approximate message passing using in-memory computing“, S. 1–8.

[4] A. Shafiee *u. a.*, „ISAAC: A Convolutional Neural Network Accelerator with In-Situ Analog Arithmetic in Crossbars“, in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, Juni 2016, S. 14–26, doi: 10.1109/ISCA.2016.12.

[5] M. Gautschi *u. a.*, „Near-Threshold RISC-V Core With DSP Extensions for Scalable IoT Endpoint Devices“, *IEEE Trans. Very Large Scale Integr. VLSI Syst.*, Bd. 25, Nr. 10, S. 2700–2713, Okt. 2017, doi: 10.1109/TVLSI.2017.2654506.

[6] Antonio Pullini, Davide Rossi, Igor Loi, Alfio Di Mauro, Luca Benini, "Mr.Wolf: A 1 GFLOP/s Energy-Proportional Parallel Ultra Low Power SoC for IoT Edge Processing", *In Proc. European Solid State Circuits Conference (ESSCIRC) 2018, 3-6 Sep 2018, Dresden*, **DOI:** 10.1109/ESSCIRC.2018.8494247