Horizon 2020 Program (2014-2020)

FET-Open Novel ideas for radically new technologies

FETOPEN-01-2018-2019-2020

WiPLASH

Architecting More than Moore – Wireless Plasticity for Massive Heterogeneous Computer Architectures [1†]

D5.2: Die-level exploration at design-time/runtime

WP5 - Multi-scale Simulation

Contractual Date of Delivery	30/09/2021
Actual Date of Delivery	29/09/2021
Deliverable Security Class	Public
Editor	Giovanni Ansaloni (EPFL)
Contributors	EPFL (Leader), IBM
Quality Assurance	Irem Boybat(IBM), Renato Negra(RWTH)

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

Document Revisions & Quality Assurance

Deliverable Number	D5.2
Deliverable Responsible	EPFL
Work Package	WP5
Main Editor	Giovanni Ansaloni

Internal Reviewers

1. Irem Boybat (IBM)
2. Renato Negra (RWTH)

Revisions

Version	Date	By	Overview
0.1	17/07/2021	Giovanni Ansaloni	Document created
0.1.2	4/08/2021	Joshua Klein, Rafael Medina	Section 3/4 drafted
0.1.3	2/09/2021	Rafael Medina	Architectural exploration finalized
0.1.4	17/09/2021	Giovanni Ansaloni	Update after internal review
1.0	29/09/2021	Giovanni Ansaloni	Minor updates, submitted version

Executive Summary

In its first part, the deliverable describes the rationale guiding the design of the WiPLASH system-level simulator. Then, it details the strategy employed to develop its novel modules, which simulate Analog In-Memory Computing (AIMC) memristive arrays and wireless in-die interconnects, respectively. These two components allow for the exploration of the performance of deep-learning applications on heterogeneous systems-on-chip featuring on-die wireless antennas. Hence, showcasing the added value provided by WP5 at this stage of the WiPLASH project. The last part of the deliverable reports an architectural exploration targeting the AlexNet benchmark, considering systems with different computation and communication capabilities.

Abbreviations and Acronyms

AIMC	Analog In-Memory Computing
BW	Bandwidth
CL	Convolutional Layer
CNN	Convolutional Neural Networks
CPU	Central Processing Unit
FCFS	First Come First Served
ISA	Instruction Set Architecture
MVM	Matrix-Vector Multiplication
RGB	Red Green Blue
RRAM	Resistive Random Access Memory
SoC	System on Chip
SPM	Scratchpad Memory
WP	Work Package

The *WiPLASH* consortium is composed by:

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

Table of Contents

List of Figures

List of Tables

1 Introduction

The multiscale simulation Work Package 5 (WP5) aims at demonstrating the performance provided by innovative computing architectures. It provides system-wide run-time execution metrics when executing complex applications, allowing the systematic exploration of the performance deriving from different settings of architectural parameters.

In the context of the overall objectives of the WiPLASH project, particular focus of WP5 is on (a) the modelling on-die and on-package wireless connection and (b) on the integration of Analog In-Memory Computing (AIMC) accelerator models.

These two aspects are interdependent, as an increase in computation capability due to acceleration may increase the pressure on CPUs-to-memory and CPUs-to-CPUs links, shifting the performance bottleneck from computation to communication. The effectiveness of hardware acceleration is hence ultimately dependent on the available interconnect bandwidth.

To co-explore these facets, in WP5 we are developing a full-system simulation infrastructure, based on gem5-X [1][2]. The simulator allows to rapidly define and run applications on hardware-accelerated and wirelessly enabled multicore systems.

Simulations center on the execution of AI applications, in particular Convolutional Neural Networks (CNNs). Such a choice is motivated by CNNs high workload in terms of both computation and communication requirements, which makes them challenging targets for our explorations as reported in D4.1. Since the main computational hotspots of CNN applications are Matrix-Vector Multiplications (MVMs), we consider dedicated AIMC tiles [3][4] to speed up these computational patterns.

Summing up, this deliverable illustrates the research and development activities undertaken in the context T5.2 and the first months of T5.3:

- We developed a module modelling inter-CPU wireless communication, integrated within the gem5-X full-system simulation environment. The wireless communication module can be employed to define systems with heterogeneous (wireless and wired) interconnects.
- We integrated the wireless communication and AIMC acceleration models (the latter being first presented in D5.1) in a unified framework.

Targeting the AlexNet [5] CNN benchmark, we illustrate a preliminary system exploration, illustrating the impact of the wireless communication bandwidth on the run-time of a multi-CPU system, with and without AIMC acceleration.

2 Enabling system co-design loops

The first objective towards which the effort in WP5 is directed, is to integrate the findings on WP3 and WP4 from a more high-level viewpoint. This stance allows the evaluation of run-time metrics of entire systems, executing large applications. In this context, we are focusing both on the AIMC characterization and that of the wireless antennas and communication protocols.

The second objective is to provide valuable feedback to the other WPs of the relative benefits of different arrangements, and where performance bottlenecks can be expected in different scenarios. Such information will allow other WPs to refine their requirements, avoiding the pitfalls of under- or over-designing hardware components and/or protocols. Indeed, it will provide important inputs for the design of the AIMC accelerator models (T4.1) and for the development of wireless MAC protocols (T3.3) and networks (T3.4).

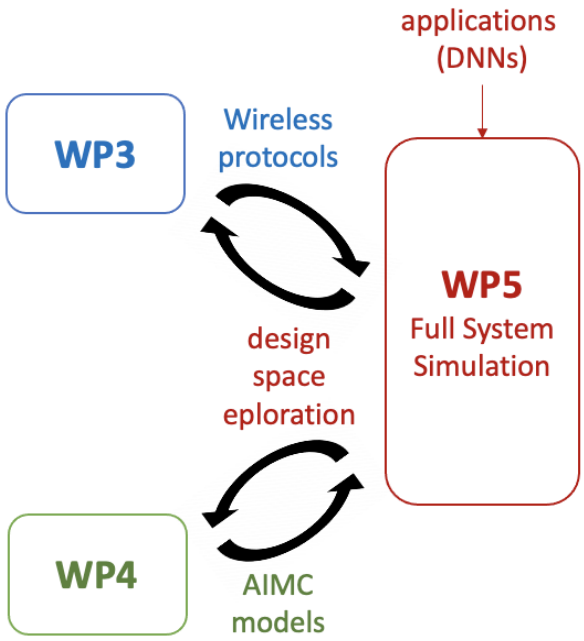

Figure 1: Implemented co-design loops.

Feedback can be provided both with high-level metrics summarizing the run-time, packet collisions etc. of an execution on a target system. Indeed, Figure 2 shows a graph illustrating the wireless transmission bandwidth requirements originating from a multi-CPU system executing AlexNet inference, where the different convolutional, max-pooling and linear layers are mapped on CPUs 1-6, while CPU 0 is used to orchestrate the execution.

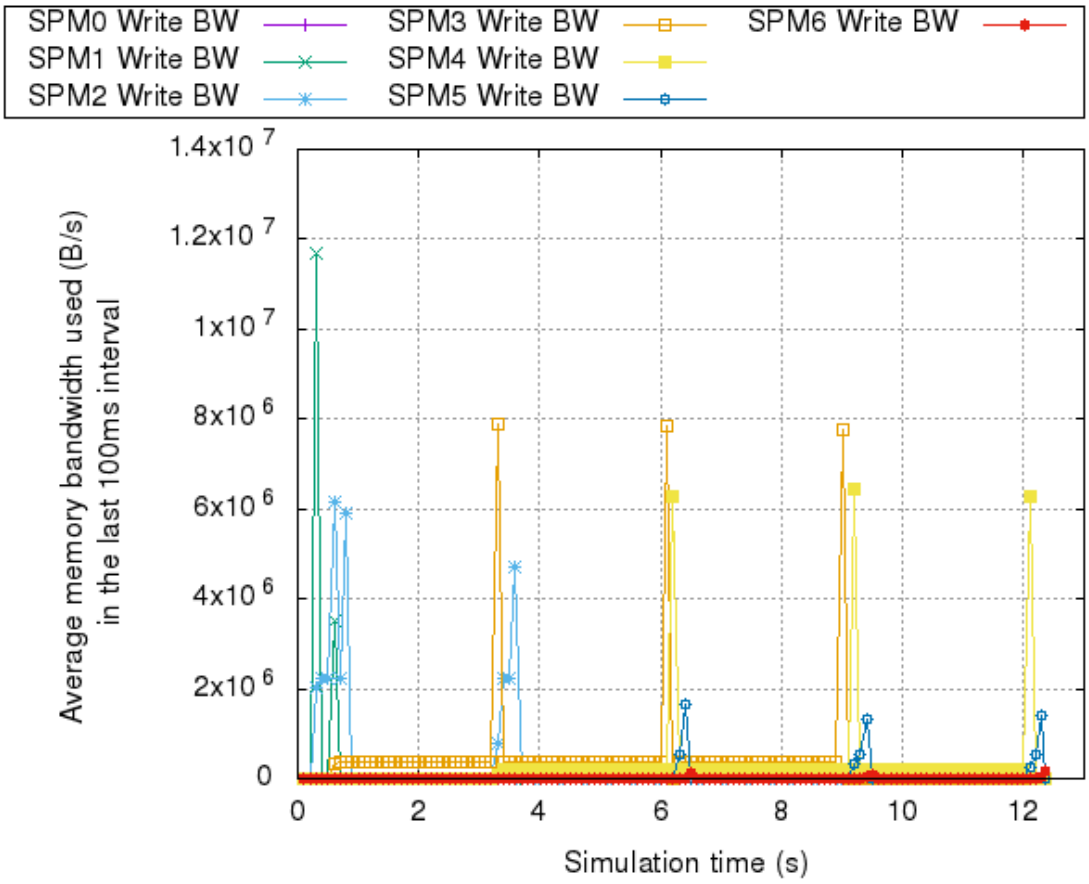

Figure 2: Example of aggregated statistics: wireless bandwidth for different CPUs.

Detailed execution traces can also be obtained, for example illustrating time annotations for each wireless transmission among couples of CPUs. Such traces will complement the ones provided in WP4. The format of the traces is shown in Figure 3, which reports a brief excerpt. They are obtained by employing gem5 components named Communication Monitors [7]. The obtained data is then filtered to derive the operations of interest, for example wireless transmission operations in Figure 3.

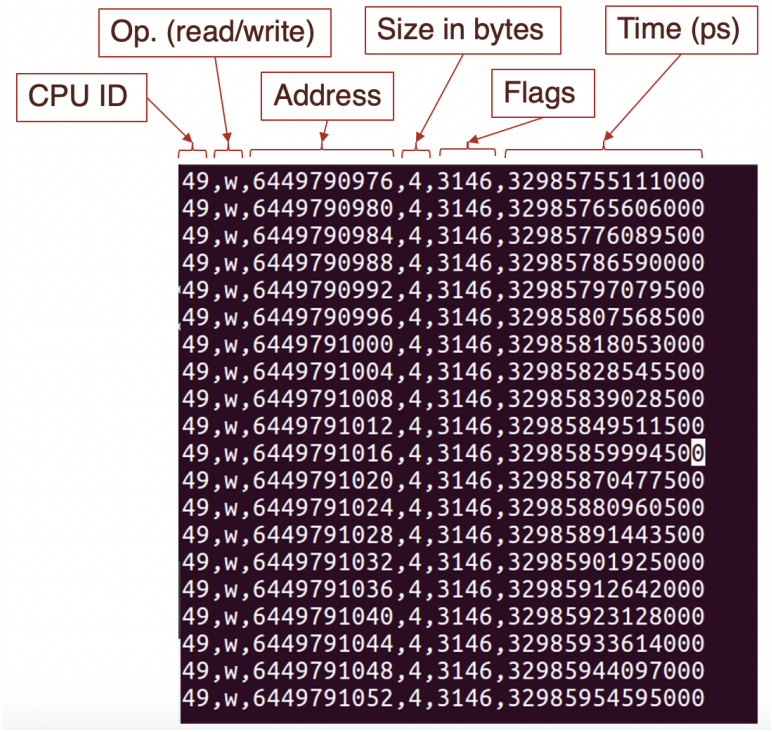

Figure 3: Excerpt of detailed execution trace.

We aim at realizing the above-mentioned objectives while maintaining a high degree of configurability in the development of system simulation components. Indeed, both the AIMC and the wireless connection components are parameterizable across multiple dimensions *e.g.*, regarding the size of the AIMC accelerators and the bandwidth of wireless communication links. Parameters can then be easily swept by modifying gem5-X system configurations, allowing for rapid and automated design space explorations.

We aimed at supporting novel features in self-contained modules, hiding their implementation and only exposing clear and simple interfaces to system hierarchies. In this way, we minimized the effort required to define wirelessly communicating and AIMC-accelerated multi-CPU systems, as all standard gem5 components such as CPUs, caches, main memories *etc.* can be reused without modifications. Moreover, we allowed for easy upgrades of the supported features of the developed components, such as the integration of more complex or detailed protocols mirroring the progress in WP3.

3 Modelling wireless communication links in gem5-X

We encapsulated the model of wireless transmission channels and protocols as a self-contained gem5-X component. Hence, as discussed in the previous section, wireless Systems-on-Chip (SoCs) can be defined while reusing existing gem5 components for processors, buses, memories *etc*. [6]. Indeed, adding a wireless channel to a design simply consists in connecting, in the system configuration file, the system components (e.g. CPUs) that communicate wirelessly through the channel module.

The code snippet below exemplifies the instantiation of a wireless channel. It contains the component declaration, which refers to the header file as an entry point to the module implementation. Moreover, the snippet reports the code lines instantiating the channel component with the desired parameters, and the ones connecting the created components with the CPUs and Scratchpad Memories (SPMs), as detailed further in this section.

```
# Wireless Channel declaration
  class WirelessChannel(MemObject):
      type = 'WirelessChannel'
      cxx_header = "mem/wireless_channel.hh"
 ...

# Instantiation of a Wireless Channel
  system.wl_channel =
   WirelessChannel.WirelessChannel(
       bandwidth = options.wireless_bandwidth,
         <...other parameters...>
     )

# Connecting the channel to the crossbars in the CPU and SPM
# sides
    system.cpu_channel_bus.master = system.wl_channel.slave
    system.channel_spm_bus.slave = system.wl_channel.master
```

Excerpt of system configuration file instantiating a wireless channel among multiple CPUs.

Our focus is to reproduce two defining characteristics of CPU-to-CPU wireless links:

- limitations in bandwidth
- collisions when multiple CPUs try to transmit data through the same channel at the same time.

From a high-level perspective, the module employs three main components: buffers (realized as SPMs) are employed to store the data being received by each processor, while crossbars connect buffers, CPUs, and the wireless channel model, as illustrated in Figure 4. Such implementation allows decoupling the content of the transmitted data, the connectivity among CPUs, and the characteristic of the wireless channel. In particular, the latter is only described in the wireless channel

block, minimizing the effort required to extend our implementation and support different protocol designs [8] [9]. As an example, while at present we only support retransmission-based mechanisms, our approach can easily be extended to token-based ones.

In more detail, in order to model the wireless transmission, we assign a buffer to each of the connected CPUs. A write operation from a CPU to the buffer of a different CPU is interpreted as a wireless transmission, and thus the limitations of bandwidth and packet collisions are considered for the communication. On the other hand, we allow direct, non-wireless read and write from a CPU to its own buffer, as these are interpreted as local accesses. All CPU-buffers data traffic goes through the wireless channel model. The module will then decide whether the access is local or remote, depending on the CPU that initiated it and the access address.

The wireless channel module (see again Figure 4) presents two ports, facing distinct crossbars[2], one for the CPUs side and the other for the buffers side. Crossbars are idealized (*i.e.*, they have no associated delay), in order not to influence the timing behavior of the wireless communication.

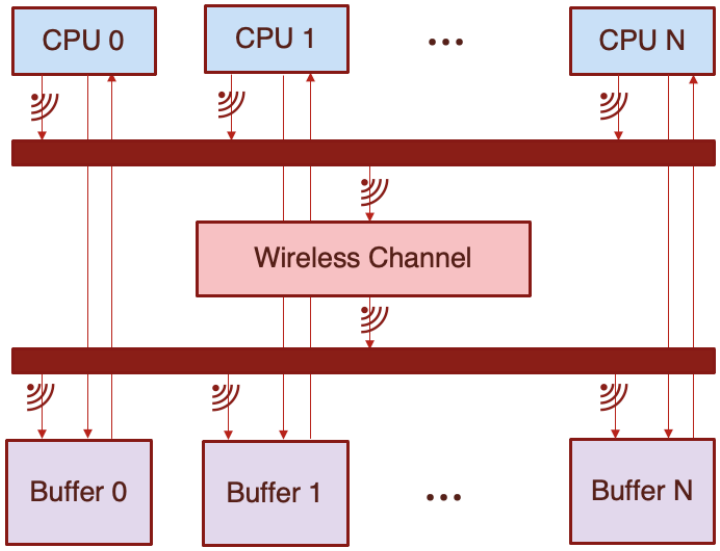

Figure 4: Block scheme and architecture of the developed wireless model gem5-X component.

Embedded in the wireless channel module is a transmitter unit which serves the incoming write request packets and enables the modelling of bandwidth and collisions. The transmitter is busy for a time corresponding to the available transmission bandwidth, attempting to forward data packets to their destination. If there is no collision, the release of the transmitter signals the successful completion of the wireless transmission. The transmission time (T) of the forwarding function is set accordingly to the size of the packet and the modelled bandwidth (BW) of the wireless channel, according to the following formula:

[2] Crossbar components are used in gem5 to model many-to-many connections

$$T = Size\ (packet)\ /\ BW$$

The transmitter will reject packets if it is busy. This can be interpreted as:

1. Reaching the bandwidth limit of the channel, if the CPU originating the rejected packet is the same as the one transmitting.
2. A collision of packets, if the source of the new packet is different to the one of the packet being transmitted. This is equivalent to two or more different CPUs trying to wirelessly transmit at the same time.

In both cases, the CPU originating the rejected packet is stalled, and is signaled to retry the transmission once the current transmission has been served. This strategy therefore implements a retransmission-based protocol.

The ownership of the transmitter unit is obtained by CPUs in a First-Come First-Served (FCFS) fashion. In case the transmitter is idle and two CPUs try to transmit in the same cycle, the order of ownership is sorted following gem5-X's order of core execution, in which the CPUs are identified from lower to the higher indices.

As depicted in Figure 5, to model the timing behavior of collisions, we delay the write responses for a random time in a (parameterizable) window, which results in colliding transmissions to incur in random delays. As again shown in Figure 5, re-transmission attempts can also collide. Hence, we repeat the above-mentioned strategy iteratively increasing the window size, following an exponential backoff approach.

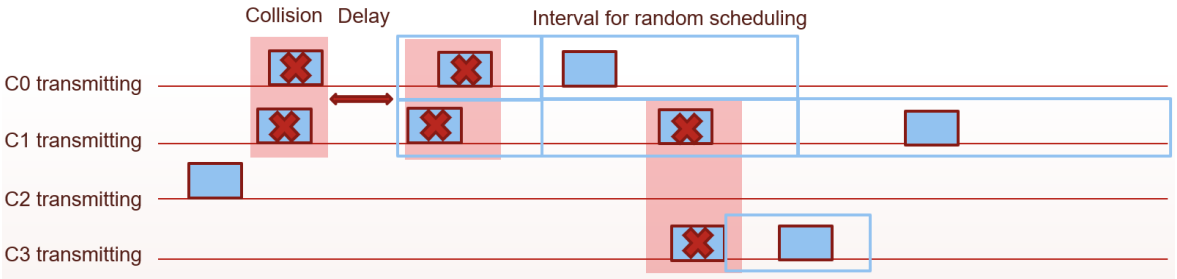

Figure 5: Timing diagram of collision modelling.

Collisions are detected by checking if a) a CPU is trying to use the transmitter while it is owned by other CPUs, or b) a CPU is trying to use the transmitter while previous collisions are still being resolved. In both cases, involved packets are delayed proportionally to the total number of ongoing collisions:

$$delay = (n\text{-}1) \times W$$

where n is the number of collisions to be addressed and W is the minimum window size, which is enough waiting time for all the colliding transmissions to have been served by the transmitter. After the delay, each response packet of the colliding transmissions is assigned a random time to be forwarded to the source CPU. When the assigned transmission time arrives, we check for overlaps with the other

scheduled retransmission. If an overlap occurs, the involved packets are reassigned at random retransmission times; otherwise, the transmission of the packet successfully ends. The random transmission time is chosen within an interval whose size is decided via the exponential backoff algorithm. The minimum interval size (*W*) and the base of the exponent for backoff can be configured. In our first experiments, we fixed the retry window values to:

$(2^c - 1) \times W$, where c is the number of retries incurred by a packet.

This approach makes colliding more unlikely the more retransmissions occur.

In Section 5 we explore the performance of different CPU-to-CPU wireless bandwidths by targeting an 8-CPU system in which all cores are connected to each other through a wireless channel. We use per-core Scratchpad Memories (SPMs) connected to the buffer port of the wireless channel model to enable inter-CPU transmissions. Applications index memory locations in the SPM address ranges (as determined by the system configuration script) in order to write and read to local and remote SPMs. A write to a remote SPMs triggers a wireless transmission.

4 Modeling AIMC tiles in the target architecture

Analog In-Memory Computing (AIMC) accelerators are formed by a series of vertical and horizontal lines in a pattern carrying input and output signals, as depicted in Figure 6. At each intersection, programmable Resistive RAMs (RRAMs) connect horizontal and vertical wires. The current at the outputs of the accelerator is a linear combination of the voltages of the inputs, weighted according to the resistance values. Analog-to-digital converters are employed to sense these currents. Hence, a *N*-Input, *M*-output AIMC embeds *N*x*M* contacts (programmable resistances). At each execution it computes at once the Matrix-Vector Multiplication (MVM) of the vector of the inputs with the matrix encoded in the resistances.

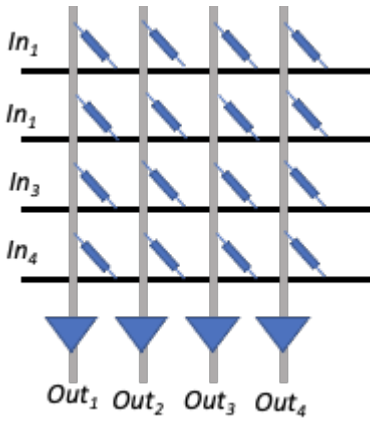

Figure 6: Idealized schematic of an AIMC architecture.

The design and integration of AIMC tiles are active and promising research topics [10]. AIMC is particularly suited for the acceleration of AI applications such as CNNs, as these are strongly relying on MVM operations. An advanced prototype AIMC implementation was recently demonstrated by IBM research[3].

We firstly introduced our gem5-X AIMC model in D5.1. We now showcase in Section 5 its use in the context of a multi-core wireless system. In our gem5-X implementation, we model AIMC tiles as peripheral input/output components, which can be embedded in single- or multi-CPU systems. Moreover, one or multiple AIMC tiles can be instantiated for each CPU. AIMC models can be parametrized through the system configuration file in terms of width and height, which correspond to the desired number of inputs and outputs, all represented as 8-bit integers.

Below is an excerpt of a gem5-X example configuration script specifying and integrating an AIMC accelerator. The accelerator is first declared as an available system component, whose characteristics are defined in the “AIMC.hh” header file. Instances (*e.g.* “aimc”) are then instantiated, and they are finally connected using the desired links (“bus.master” in the example).

```
# AIMC tile declaration
 class AnalogComputationalMemory(BasicPioDevice):
    type = 'AnalogInMemoryComputing'
    cxx_header = "dev/arm/AIMC.hh"
    ...

# Instantiation of an AIMC accelerator
 aimc = AnalogComputationalMemory()

# Connecting the AIMC tile on the bus.
 self.aimc.pio= bus.master
```

Excerpt of system configuration file instantiating an AIMC accelerator module.

CPUs and AIMC tiles are tightly coupled, so that a CPU can only interface with its local AIMC tile, governing them via special opcodes extending the ARMv8 ISA. Such instructions allow managing the transfer of inputs and outputs to/from AIMC tiles, as well as to program the resistances encoding the weights matrix. It has to be noted that changing the resistance values is a timing and energy intensive operation, both because of the electrical characteristics of resistive memories and because *N*x*M* connections are present in the AIMC tiles. This motivates our choice, in Section 5, of not modifying them during the benchmark execution, mapping instead different matrix-vector-multiplications (realizing different model layers) in different AIMCs tiles.

The 5 new custom instructions added to the ARMv8 ISA are termed CM_QUEUE, CM_DEQUEUE, CM_PROCESS, CM_P_W, and CM_P_R. CM_QUEUE and CM_DEQUEUE queues 8-bit inputs and dequeue 8-bit outputs from the input and output memories of the AIMC tile, respectively. CM_PROCESS signals the AIMC tiles to perform the constant-time MAC operations using the queued input memory values, and then store the output in the output memory. This operation also clears the input memory. Finally, the CM_P_R and CM_P_W instructions read and write parameters directly to the AIMC tile, respectively. Parameter writes are used to program the AIMC tile while parameter reads are only used for debugging purposes.

The usual flow of these instructions is as follows:

1. During the programming/setup phase of an application, CM_P_W is called to write weights/parameters into the AIMC tile.
2. Inputs are queued into the input memory via the CM_QUEUE instruction. The instruction accepts a 32-bit value, so that up to four 8-bit inputs can be queued into the input memory with one instruction call. It has a latency of 1 ns per call.
3. Once the input memory is set, CM_PROCESS is called to perform the MVM, clear in the input memory, and store the output in output memory. Only one instruction invocation is needed to compute the entire matrix-vector multiplication, which has a latency of 100 ns per call

4. Finally, the CM_DEQUEUE instruction is called to fetch the outputs in a similar manner to CM_QUEUE (*i.e.*, up to four 8-bit outputs are fetched from output memory).

To ease the usage of these instructions at the application level, we developed a dedicated C++ library, named aimclib. The library is a collection of header files, but its use in software applications only requires including the top-most header.

It contains wrappers for the instruction intrinsics as well as helper functions to handle the queueing, dequeuing, and the mapping of matrices to the AIMC tile. Also present are utilities for scaling inputs/outputs when not using int8_t types, performing convolutions, and more. The library also has support for manipulating data structures from the Eigen library [11].

As an example, below is a C++ pseudocode snippet for writing to the AIMC accelerator, queueing an input array, performing a single MVM operation, and dequeueing into an output array:

```
#include “aimclib.hh”

int main(int argc, char * argv[]) {
        …

        // Weights to be placed in the AIMC tile
        // with x/y dimension of N.
        int8_t ** aimcMatrix = = { { … }, …};

        // Mapping weights to the AIMC tile
        // with x, y offset of 1, 1 using aimclib.
        mapMatrix(1, 1, N, N, aimcMatrix);

        // Input to be queued into AIMC tile input memory
        // and output array to be dequeued into.
        int8_t * input = { … }, output = new int8_t[N];

        // Queue input array into the AIMC tile input memory
        // using aimclib.
        queueVector(sizeof(input) / sizeof(input[0]), input);

        // Perform MVM using aimclib.
        aimcProess();

        // Dequeue output memory contents into output array
        // using aimclib.
        dequeueVector(N, output);

        return 0;
}
```

Example application code interfacing with an AIMC accelerator using aimclib.

5 System Architecture exploration

5.1 Experimental setup

To perform a first evaluation of the performance of wireless-enabled multi-CPU SoC varying the transmission bandwidth, we considered AlexNet [5] as a first application target. The application performs the inference of 10 possible objects on three images from the CIFAR10 dataset [12]. Inputs are of dimensions 32x32x3 (32x32 images in RGB format). The outputs contain, for each image, one score for each of the 10 classes.

The AlexNet CNN is composed of 9 layers,

- 5 Convolutional layers
- 2 Max pooling layers
- 2 Linear layers

The image below summarizes the application structure. It also shows how layers are mapped to the multi-CPU system, whose characteristics are also detailed in this section.

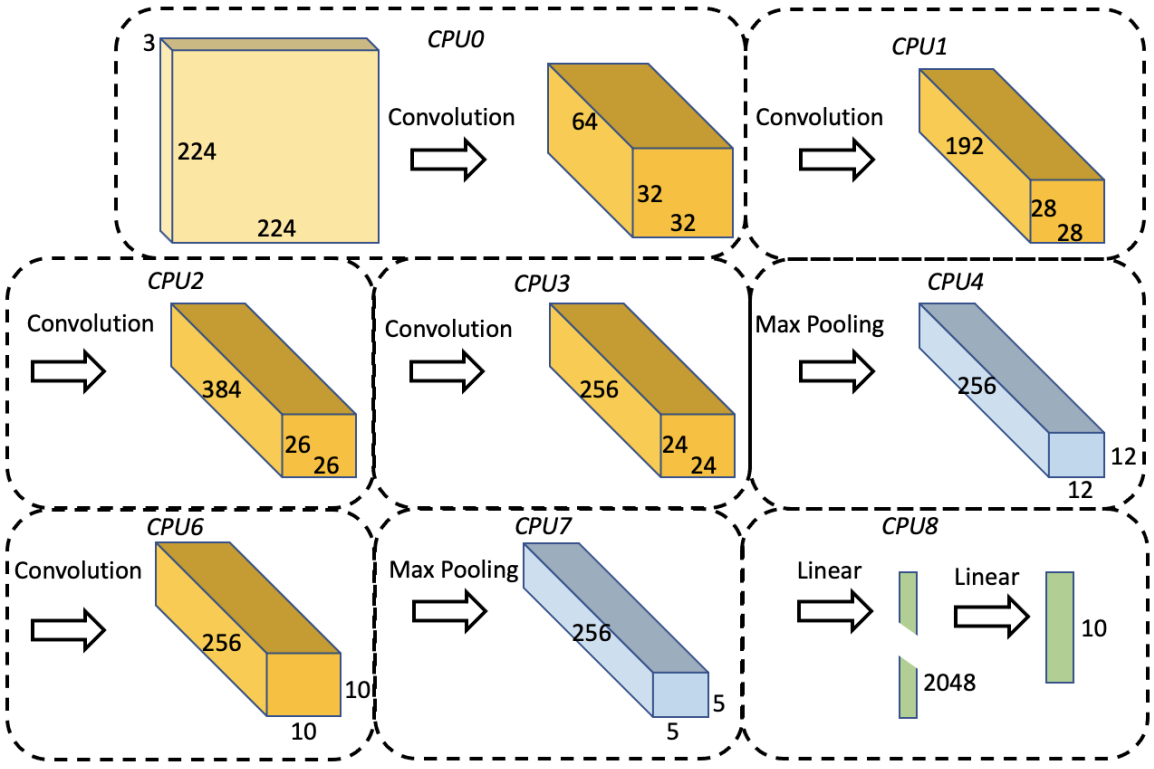

Figure 7: AlexNet layers characteristics and their mapping to CPUs in the considered benchmark application.

Hardware acceleration with AIMC tiles has been considered for the convolutional layers, since those are the most computationally demanding. Similarly to the work of [13], we mapped all the coefficients of a convolutional filter in a single array

column of the AIMC tile, so that columns host the weights for different filters. In this way, weight values, encoded in the memory elements of the AIMC tile, are static after initialization at run-time, avoiding the need for costly reprogramming operations. Each accelerator invocation is performed with a proper slice of the input activations, in order to produce (*1 × 1 × N_out*) output values, where *N_out* is the number of output channels.

More details on activation and weight size of convolutional layers are provided in the table below.

Table 1: Characteristics of the Convolutional layers in the benchmark application.

	Conv. Layer 1 (CL1)	**Conv. Layer 2 (CL2)**	**Conv. Layer 3 (CL3)**	**Conv. Layer 4 (CL4)**	**Conv. Layer 5 (CL5)**
Output Depth	64	192	384	256	256
Output Width	30	28	26	24	10
Output Height	30	28	26	24	10
Kernel size	3x3	3x3	3x3	3x3	3x3
Stride	1	1	1	1	1
Padding	0	0	0	0	0

To maximize the application parallelism, the CNN execution is pipelined at the slice level: the cores are able to process parallelly different horizontal slices of the activations, as soon as they have received them from previous cores. As an example, since the CL3 has a kernel size of 3, its first output row is computed by CPU2 as soon as the first three rows of activation data are available from CL2, produced by CPU1.

The experimental vehicle for the performed exploration is an 8-CPUs system, with the following characteristics

- CPUs: 8x ARMv8 Minor, operating at 2.0 GHz
- L1 Instruction Cache: 8x private caches, 32 kB
- L1 Data Cache: 8x private caches, 32 kB
- L2 Cache: 512 kB, shared
- Main memory: 4 GB DDR4 RAM
- Operating system: Ubuntu LTS 16.04

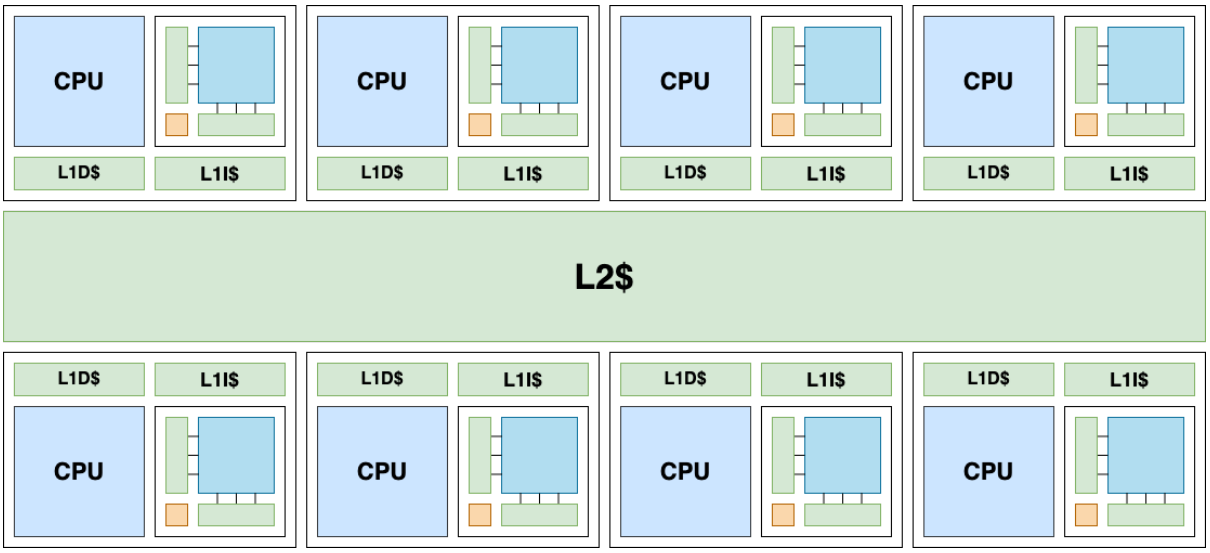

Figure 8: Block scheme of the simulated architecture, featuring 8 CPUs, private L1 and shared L2 caches.
Each CPUs interfaces a tightly coupled AIMC accelerator

We modelled a single inter-CPUs wireless channel and considered bandwidths of 100 Gbps, 30 Gbps, 10 Gbps, 3 Gbps, 1 Gbps, 0.3 Gbps, and 0.1 Gbps. Such bandwidths sustained the transmission of activations from one layer to the next.

AIMC hardware acceleration employs per-core AIMC tiles with 3456 Rows and 384 Columns, which allows mapping the matrix-vector multiplication required for the computation of each AlexNet convolutional layer in an AIMC module.

For comparison purposes, we also modelled less capable CPU-only baseline configurations, which did not feature AIMC acceleration.

5.2 Exploration outcomes

Table 2 reports a run-time comparison, for various wireless bandwidth constraints, of the AlexNet application executing with and without AIMC acceleration. As expected, hardware accelerated systems effectively speed-up execution, as they can perform MVM in linear time.

Moreover, results confirm the intuition that, by increasing the computational efficiency, hardware acceleration puts an increased strain on computational resources. Indeed, when not employing AIMC accelerators no measured improvement on run-time was detected when increasing the wireless bandwidth past 0.3 GBps. Conversely, when AIMC tiles are integrated tangible run-time reductions are found by increasing the available bandwidth up to 3 Gbps.

Table 2: Run-time of the AlexNet benchmark on the simulated system, with and without AIMC acceleration, for different wireless channel bandwidths.

Run-time (s)	100 Gbps	30 Gbps	10 Gbps	3 Gbps	1 Gbps	0.3 Gbps	0.1 Gbps
w/o AIMC	14.9	14.9	14.9	14.9	14.9	14.9	15.0
with AIMC	0.357	0.357	0.359	0.385	0.473	1.41	4.31

Smaller available bandwidths indeed results in much higher collision rates among wireless data transmissions, resulting in overall run-time slowdowns. Further detailing our performance assessment, collisions are reported in Table 3 and 4 for systems with and without AIMC accelerators, respectively. In each case, we illustrate the collisions (and the collision rates) reported by each core, as well as the aggregated results. A tangible discontinuity in collision rates can be noticed when reducing the available bandwidth from 0.3 to 0.1 Gbps in the nonaccelerated case (Table 3) , and from 10 to 3 Gbps in the accelerated one (Table 4).

Table 3: Packet collision on the simulated system without AIMC acceleration, for different wireless channel bandwidths.

Collisions (%)	100 Gbps	30 Gbps	10 Gbps	3 Gbps	1 Gbps	0.3 Gbps	0.1 Gbps
Core 0	0 (0.0%)	0 (0.0%)	52 (0.030%)	389 (0.22%)	1238 (0.70%)	3210 (1.8%)	6982 (4.0%)
Core 1	0 (0.0%)	0 (0.0%)	101 (0.022%)	395 (0.086%)	992 (0.22%)	3856 (0.84%)	20003 (4.4%)
Core 2	0 (0.0%)	0 (0.0%)	74 (0.009%)	713 (0.091%)	1556 (0.20%)	3907 (0.50%)	14945 (1.9%)
Core 3	0 (0.0%)	1 (0.0%)	99 (0.022%)	789 (0.18%)	1050 (0.24%)	3300 (0.74%)	12467 (2.8%)
Core 4	0 (0.0%)	0 (0.0%)	44 (0.039%)	163 (0.14%)	366 (0.32%)	532 (0.47%)	873 (0.77%)
Core 5	0 (0.0%)	0 (0.0%)	8 (0.010%)	76 (0.098%)	293 (0.38%)	870 (1.1%)	3937 (5.1%)

Core 6	0 (0.0%)	1 (0.010%)	8 (0.081%)	16 (0.16%)	81 (0.82%)	99 (1.0%)	116 (1.2%)
Core 7	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)
Total	**0 (0.0%)**	**2 (0.0%)**	**386 (0.019%)**	**2541 (0.12%)**	**5576 (0.27%)**	**15774 (0.76%)**	**59323 (2.9%)**

Table 4: Packet collision on the AIMC-accelerated simulated system for different wireless channel bandwidths.

Collisions (%)	100 Gbps	30 Gbps	10 Gbps	3 Gbps	1 Gbps	0.3 Gbps	0.1 Gbps
Core 0	0 (0.0%)	56 (0.032%)	58733 (33%)	119423 (68%)	155996 (89%)	163520 (93%)	178065 (101%)
Core 1	0 (0.0%)	118 (0.026%)	109449 (24%)	213024 (47%)	173066 (38%)	197703 (43%)	240758 (53%)
Core 2	0 (0.0%)	111 (0.014%)	88224 (11%)	263536 (34%)	267522 (34%)	407721 (52%)	515433 (66%)
Core 3	0 (0.0%)	114 (0.026%)	102617 (23%)	309487 (69%)	326081 (73%)	433698 (97%)	484556 (109%)
Core 4	0 (0.0%)	0 (0.0%)	25360 (22%)	90818 (80%)	90529 (80%)	106287 (94%)	120728 (107%)
Core 5	0 (0.0%)	0 (0.0%)	18635 (24%)	63313 (82%)	71768 (92%)	75196 (97%)	86405 (111%)
Core 6	0 (0.0%)	324 (3.3%)	3482 (35%)	5327 (54%)	7272 (73%)	8206 (83%)	10044 (101%)
Core 7	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	2 (67%)	2 (67%)	2 (67%)
Total	**0 (0.0%)**	**723 (0.035%)**	**406500 (20%)**	**1064928 (52%)**	**1092236 (53%)**	**1392333 (68%)**	**1635991 (79%)**

6 Conclusions and Perspectives

This deliverable marks the establishment of a novel full-system simulation infrastructure extending gem5-X, able to model wireless-enabled and hardware-accelerated SoCs.

The availability of such infrastructure, besides being a major undertaking on its own, also has important consequences for the project as a whole. Indeed, the system simulator allows the application-wide assessment of the benefits deriving from different implementations of on-die and on-package wireless components, providing valuable feedback to other WPs.

We are at present reaping the benefits deriving from this implementation effort. In the context of the ongoing project task T5.3, we are now investigating further benchmarks and wireless transmission mechanisms (e.g., retransmission-based versus token-passing). Moreover, we are broadening our scope, considering further dimensions and opportunities offered by CPU-to-CPU wireless links, including but not limited to thermal balancing considerations.

Bibliography

[1] Qureshi YM, Simon WA, Zapater M, Atienza D, Olcoz K. Gem5-X: A Gem5-based system level simulation framework to optimize many-core platforms. In2019 Spring Simulation Conference (SpringSim) 2019 Apr 29 (pp. 1-12). IEEE.

[2] Qureshi, Yasir Mahmood, et al. Gem5-X: A Many-Core Heterogeneous Simulation Platform for Architectural Exploration and Optimization. ACM Transactions on Architecture and Code Optimization (TACO). 2021.

[3] Burr GW, Shelby RM, Sidler S, Di Nolfo C, Jang J, Boybat I, Shenoy RS, Narayanan P, Virwani K, Giacometti EU, Kurdi BN. Experimental demonstration and tolerancing of a large-scale neural network (165 000 synapses) using phase-change memory as the synaptic weight element. IEEE Transactions on Electron Devices. 2015 Jul 7;62(11):3498-507.

[4] Ielmini, Daniele, and H-S. Philip Wong. In-memory computing with resistive switching devices. *Nature Electronics* 1.6 (2018): 333-343.

[5] Krizhevsky A, Sutskever I, Hinton GE. Imagenet classification with deep convolutional neural networks. Advances in neural information processing systems. 2012;25:1097-105.

[6] Binkert N, Beckmann B, Black G, Reinhardt SK, Saidi A, Basu A, Hestness J, Hower DR, Krishna T, Sardashti S, Sen R. The gem5 simulator. ACM SIGARCH computer architecture news. 2011 Aug 31;39(2):1-7.

[7] gem5 communication monitors. Available online: http://pages.cs.wisc.edu/~swilson/gem5-docs/classCommMonitor.html

[8] Abadal S, Cabellos-Aparicio A, Alarcón E, Torrellas J. WiSync: An architecture for fast synchronization through on-chip wireless communication. ACM SIGPLAN Notices. 2016 Mar 25;51(4):3-17.

[9] Fernando V, Franques A, Abadal S, Misailovic S, Torrellas J. Replica: A wireless manycore for communication-intensive and approximate data. InProceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems 2019 Apr 4 (pp. 849-863).

[10] Bavikadi S, Sutradhar PR, Khasawneh KN, Ganguly A, Pudukotai Dinakarrao SM. A review of in-memory computing architectures for machine learning applications. InProceedings of the 2020 on Great Lakes Symposium on VLSI 2020 Sep 7 (pp. 89-94).

[11] Eigen library. Available online: https://eigen.tuxfamily.org/

[12] Krizhevsky A, Hinton G. Learning multiple layers of features from tiny images. 2009.

[13] Yakopcic C, Alom MZ, Taha TM. Extremely parallel memristor crossbar architecture for convolutional neural network implementation. In2017 International Joint Conference on Neural Networks (IJCNN) 2017 May 14 (pp. 1696-1703). IEEE.