RV-Across: An Associative Processing Simulator

Jonathas Silveira[1], **Isaías Felzmann**[1], **João Fabrício Filho**[1,2], **Lucas Wanner**[1]

[1]Instituto de Computação – Universidade Estadual de Campinas

[2]Universidade Tecnológica Federal do Paraná – Câmpus Campo Mourão

jonathas.silveira@students.ic.unicamp.br

Abstract. *Associative Processing provides high-performance and energy-efficient parallel computation using a Content-Addressable Memory (CAM). Emerging big data applications can be significantly sped-up by Associative Processing, but validation and evaluation are key challenges. We present RV-Across, a RISC-V Associative Processing Simulator for testing, validation, and modeling associative operations. RV-Across eases the design of associative and near-memory processing architectures by offering interfaces to both building new operations and providing high-level experimentation. Our simulator records memory and registers states of each associative operation pass, giving the user visibility and control over the simulation. The user can employ the simulation statistics provided by RV-Across to compute performance and energy metrics. RV-Across implements common associative operations and provides a framework to allow for easy extension. We show how the simulator works by experimenting with different scenarios for associative operations with three applications that test the functionality of logic and arithmetic computations: matrix multiply, checksum, and bitcount. Our results highlight the direct relation between the data length and potential performance improvement of associative processing in comparison to regular CPU serial and parallel operation. In case of matrix multiplication, the speed-up increases linearly with matrices dimension, achieving 8X for 200x200 bytes matrices and overcoming parallel execution in an 8-core CPU.*

1. Introduction

High-performance computation and energy efficiency are fundamental in current computer systems due to the recent growth in the development of applications that require large data processing, such as Neural Network, Image Processing, and DNA [Kim et al. 2018, Gupta et al. 2019, Mutlu et al. 2019]. These applications highlight how the bottleneck between memory and CPU is a barrier to achieve better performance. Moreover, data movement consumes a significant amount of the overall system energy. In Google Docs scrolling, for example, moving data represents more than 30% of energy consumption [Boroumand et al. 2018], and memory can account for up to 41% of the energy consumption of an entire server system [Lefurgy et al. 2003].

This work was supported by the São Paulo Research Foundation (FAPESP) grant #2018/24177-0; the National Council for Scientific and Technological Development - Brazil (CNPq) grants #404261/2016-7 and #438445/2018-0; and the Coordination for the Improvement of Higher Education Personnel - Brazil (CAPES) - Finance Code 001.

A common alternative to improve performance is to adopt separate processing using CPUs, GPUs, and accelerators, which still requires data movement. Furthermore, with the end of Moore's Law and Dennard Scaling, academy and industry have an increased interest in Processing in Memory (PIM) [Mutlu et al. 2019]. PIM is an approach that aims to take computing into memory, reducing data movement and overcoming the Von-Neumann bottleneck. Processing directly in memory reduces memory access time by mitigating the physical distance and increasing the bandwidth between CPU and memory. PIM also eliminates load and store cycles, increasing energy efficiency. In this paradigm, an operation can be executed on all words in memory in parallel, in a Single Instruction Multiple Data (SIMD) approach, thus the execution time is fixed for any data length. PIM has been used to accelerate the processing of DNA, Neural Networks, and Graphs [Nai et al. 2017, Dai et al. 2019, Gupta et al. 2019, Kim et al. 2018].

Associative Processing, a PIM approach, performs in-memory parallel, logical, and arithmetic operations using lookup tables, special registers, and a Content-Addressable Memory (CAM). Recent advances in Non-Volatile Memories (NVMs) reduced the cost of implementing associative processing and attracted interest to this research area [Kaplan et al. 2017, Yantir et al. 2018, Imani et al. 2018]. However, the lack of a flexible simulation infrastructure for test and validation of in-memory operations is still an obstacle to enable its adoption [Mutlu et al. 2019].

In this work, we present RV-Across (RISC-V Associative Processing Simulator), a high-level simulator for design and validation of in-memory operations, built as an extension of the Spike reference RISC-V ISA Simulator. RV-Across provides a framework to extend, implement, and test PIM operations based on RISC-V custom extensions and instructions. Furthermore, RV-Across generates a step-by-step log of the simulation for enhanced user control. The simulator counts and logs events of comparison, writing, match, and mismatch in associative processing. These statistics are offered to the user as a means to calculate latency and energy [Yantir et al. 2018]. Our simulator makes easier to develop operations using PIM and, consequently, helps with its adoption.

The main contributions we present in this work are:

- an extensible simulation tool that enhances associative processing evaluation;
- an architectural model to interface PIM operations with an Associative Processor;
- a performance analysis of applications in the Associative Processing environment.

We show, in this work, the structure and design flow of the simulator, and how to implement new operations and evaluate applications. Our experiments show how the simulator works using two code generation and a matrix multiplication kernels. In a serial system model, all the cases resulted in fewer accesses to memory and consequently higher performance. For matrix multiplication, associative processing saves up to 61% cycles and 71% load/store operations. When considering a multicore scenario, the associative processing model achieves up to 8x of speed-up on matrix multiplication, overcoming an 8-core CPU in the 200x200 bytes matrices computation. RV-Across offers the ability of testing and evaluating in-memory operations in an easily extensible simulation tool, demonstrating how PIM affects the performance of applications on different scenarios.

2. Related Work

Processing In Memory (PIM) is an approach that increases hardware performance in terms of bandwidth (between CPU and memory), latency, and power consumption by performing computation into memory, where the data resides. The PIM concept is gaining attention due to the advancement of two technologies: 3D Memories and NVMs. In 3D memories, logical operations within the memory are already embedded, which can save energy by avoiding data movement [Santos et al. 2018, Yang et al. 2019, Nai et al. 2017, Dai et al. 2019, Zhang et al. 2014]. The advances of NVM allow lowering the implementation cost of associative processing [Kaplan et al. 2017, Yavits et al. 2018].

The interest in implementing PIM has increased in recent years, as has the construction of tools for simulating operations in memory. Most PIM simulators use a specific type of 3D memories, the Hybrid Memory Cube (HMC). Despite the exploration of CAM and HMC simulators that support full-system simulation [Oliveira et al. 2017, Leidel and Chen 2016, Jeon and Chung 2017, Xu et al. 2019, Paulo and Lima 2019], these are not compatible with associative algorithms.

In the literature, some simulators provide cycle-accurate PIM simulation using HMC. HMC-Sim [Leidel and Chen 2016] offers to the users an infrastructure of experiments with HMC 1.0 and 2.0, implementing a model to replace traditional thread mutexes with custom HMC mutex commands. In addition, the extension of HMC-Sim enables the users to craft Custom Memory Cube (CMC) operations and the evaluation of their efficacy through user applications. CasHMC [Jeon and Chung 2017] is a C++ simulator that allows a cycle-by-cycle simulation of every module in an HMC and generates analysis results including a bandwidth graph and statistical data. This simulator enables parallel execution of other simulators that generate memory access patterns. Clapps [Oliveira et al. 2017] is an HMC simulator that has an interface for the user to perform vector PIM operations in applications. Furthermore, Clapps is a parallel simulator that implements all of the HMC instructions.

PIMSim [Xu et al. 2019] and PIM-Gem5 [Santos et al. 2018] are PIM simulators with HMC that share many characteristics. Both simulators provide processor simulation with HMC, analyze the impact of PIM in the memory hierarchy, simulate full-system with cycles and energy counters, and are configurable. PIMSim integrates DRAMSim2, HMCSim, NVMain, and Gem5 for simulation at different levels (fast, Instrumentation-driven, and full-system simulation) and provides an interface for the user via directives. Pim-Gem5 implements PIM support in Gem5 and creates a methodology for prototyping PIM accelerators.

Khoram *et al.* [Khoram et al. 2018] developed an analytical model to analyze runtime, energy, and storage for a set of architectures, including Associative Processor. Specifically, the proposed method asymptotically evaluates the computational metrics in a specific architecture, ignoring constants and low-level factors. In NVSIM-CAM [Li et al. 2016], the NV-Sim simulator is adapted as a tool that estimates the performance, area, and energy of CAM and other types of NVMs. RV-Across proposes to approach associative processing, different from both Khoram and NV-Sim models. Yantir et al. [Yantir et al. 2018] proposed a methodology that combines approximate computing and associative processing. The authors developed an in-house simulator for associative operations. Yavits et. al [Yavits et al. 2015] also developed an associative processor in-house

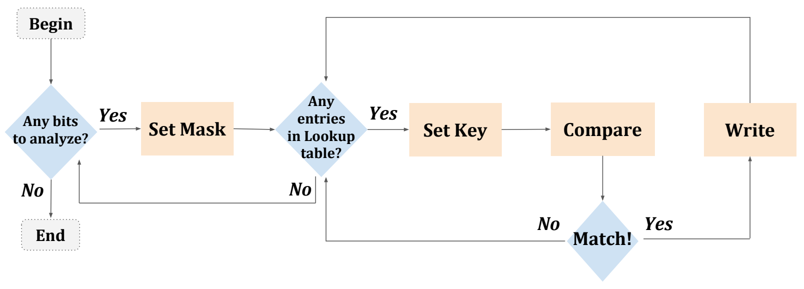


Figure 1. AP Diagram

simulator. In this simulator, the associative processor is at the last cache level. It also estimates energy based on associative processing events.

RV-Across is a simulator, focused on Associative Processor, which also allows the addition of customized operations such as HMC-Sim, implemented in C++ as CasHMC, and uses special instructions to in-memory processing as Clapps. The user, using our tool, with the events counted by the simulator will be able to extract energy statistics based on a separate model. RV-Across allows modeling extended operations and provides an interface for associative processing experiments. RV-Across uses a similar format of associative algorithm, allowing operations between vectors. However, our tool offers the freedom to build and experiment with customized operations. Our tool, as well as these simulators, delivers an interface to high-level programming (custom instructions) and enables experiments to evaluate latency and energy. RV-Across performs Associative Processing in a low-latency scratch-pad memory closely tied to the main processor, with direct access to the main memory, bypassing the cache hierarchy and avoiding memory accesses by favoring DMA bulk transfers.

3. Associative Processor

An Associative Processor (AP) is a CAM that provides additional processing capabilities, retrieving data from part of the content and operating through logical and arithmetic operations without moving data to a separate processor. In an AP, the operations can be made in several words of the memory simultaneously, reading, comparing, and writing inside the associative module without content transition. This processor needs the following components to perform computation:

- *CAM*: Associative Memory in which the data will be stored. A Content-Addressable Memory fetches the stored data from part of its content, such as a hardware implementation of a hash table.
- *Lookup Table (LUT)*: Table containing the values of the bits that will be compared and written, resulting in the operation.
- *Mask*: Register designed to select the columns that the processor will compare.
- *Key*: Register used to represent the bits of the *LUT* that are used for comparison.
- *Tag*: A bit that represents the **Match** state. This state indicates that the *LUT* comparison bits, represented by the *Key*, are the same as those of the operators. Then, the bits determined in the *LUT* in the result are written.

The Associative Processing can be summarized in three actions: selection, comparison, and writing of bits. First, the *Mask* register is set to select the columns (operands)

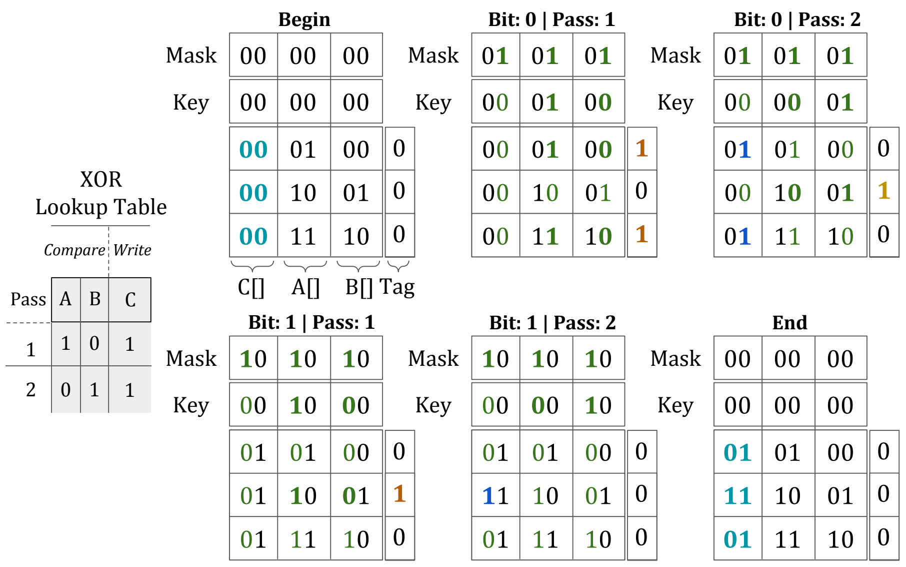


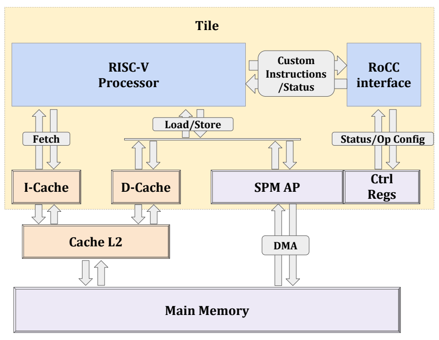
Figure 2. Xor AP operation

to be compared. Then, the *Key* is configured, representing the *LUT* comparison bits. If the bits of the *Key* and the bits selected by the *Mask* are the same, there is a **Match**, and therefore, the bit *Tag* related to the operation becomes 1. Finally, it is verified which lines had a match looking at the *Tag*. If the *Tag* is 1, the corresponding value defined in the *LUT* is written in the result. The *Mask* is set for all bits of the word, depending on the associative operation, and the *Key* for all passes of the *LUT*. Figure 1 illustrates a diagram showing this operation flow.

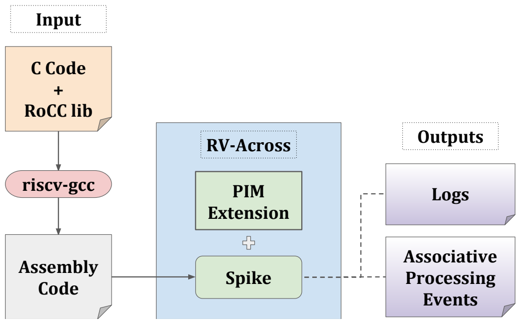
Figure 2 represents the simulation of an associative XOR between two vectors: A [1,2,3] XOR B [0,1,2]. See that the *LUT* summarizes an XOR truth table. The resulting vector C is initialized as [0,0,0], thus the entries of A and B that result in 0 are excluded from the *LUT*. Note that the *Mask* is the same for all passes computed for the same bit of the operands. Consider the first pass executed on the bit zero of the operands. The *Mask* selects the first bit of each operand, and the *Key* is read from the *LUT* for the first pass. Comparing the *Key* and the values, for the selected bit, two Matches occur (lines 1 and 3), and the *Tag* is set accordingly. The respective bits on vector C are written to 1 in the next pass (bit 0, pass 1), and the process is repeated for each bit and pass. In the end, after comparing all the passes with all the bits, the vector C will have the result of the XOR to be associative. Thus, the operation can be performed in constant time for N elements of a vector without moving data. The execution time depends, then, in the number of passes, particular to the operation being performed, and the length of each word in the data, not the size of the dataset.

4. The RV-Across PIM Simulator

RV-Across is based on an Associative Processing architectural model to support PIM. Figure 3a shows the overall architecture that our simulator represents. Inside the Tile, the processing components and instructions and data caches are located, and off the tile, the L2 cache and main memory. The main processor is a RISC-V core connected to the extension module (RoCC Interface), that provides control support for associative operations. The main core communicates with the RoCC Interface using custom instructions.



(a)



(b)

Figure 3. RV-Across architectural model (a) and design flow (b).

The RoCC Interface, when triggered, sends setup and command information to the control registers coupled to the Associative Processor, such as the addresses of the operands and output vectors and which operation to be performed. After configuring the registers, the operation is executed while the CPU waits for a response on the status of the operation. The AP contains the LUT for the associated operations and the configured algorithms. For simplification in both hardware and software simulation, the associative operations and CPU instructions do not run in parallel.

The AP works as a Scratch-Pad Memory (SPM). In addition to reading and writing, this SPM serves as a support for data processing. The AP can access the main memory using DMA (Direct Memory Access). This reduces data movement since the main core does not need to act as a bridge to transport data from the main memory to the AP. To use the AP, the user must configure the data to be processed in the SPM and activate the associative operations using custom instructions. To gain simplicity in the implementation, the SPM is small in size, since its cost in area can be high. For all test scenarios, the SPM is no larger than 128 KB.

RV-Across is coupled to the RISC-V reference ISA simulator, Spike. This section describes the interface provided between the target application and our simulator, detailing how to insert custom instructions in the application to communicate with the simulator, how to configure metrics, add and edit operations, and how the output is generated for the user. Figure 3b shows an overview of the development flow.

4.1. User interface

RISC-V RoCC instructions are used as communication interface between the associative accelerator and the main RISC-V processor. They are used in RV-Across for configure the associative operations, defining the operation code and the parameters forwarded to the accelerator. Then, for the use of associative operations, it is necessary to insert the interface instructions in the application. For this purpose, RV-Across includes a library with predefined macros, at which it is possible to enter the parameters of what operation to use, the value for the registers, and the operation code.

The code in Figure 4 shows the shape of the generic macro ROCC_INSTRUCTION, that represents the default type-R RISC-V instruction used in the RoCC extension. The first parameter, *x*, indicates which of the four RoCC custom instructions is used in the implementation. In RV-Across, a vectorized associative

```
#define ROCC_INSTRUCTION(x, rd, rs1, rs2, funct) ...
#define ADD_RVA(in_a, in_b, out, length, word_size) \
        ROCC_INSTRUCTION(2, 0, in_a, in_b, 0); \
        ROCC_INSTRUCTION(2, 0, length, out, (word_size << 3) | 1);
```

Figure 4. RoCC instructions lib

```
for(i = 0; i < DIMENSION; i++) {
  row = i * DIMENSION ;
  for(j = 0; j < DIMENSION; j++) {
    column =  j * DIMENSION;
    SET_RVA(buff, DIMENSION, *(A + row + j), 1);
    MULT_RVA(buff, B + column, buff, DIMENSION, 1);
    ADD_RVA(C + row, buff, C + row, DIMENSION, 1);
  }
}
```

Figure 5. Matrix multiply with AP

operation is triggered after two steps, in which the first configures the position of two input vectors (*rs1* and *rs2*) and the second configures the length of the vectors (*rs1*) and the position of the output vector (*rs2*). The output register *rd*, not used in this implementation, is reserved for future use. The *funct* field is used to send additional information to the AP control. If the *funct* is '0', the AP loads the pointers of the input vectors. If it is different from '0', the AP control extracts the operation that will be executed, from the 3 least significant bits and the word size from the remaining 4 most significant bits. For example, ADD_RVA (addition) is an associative operation that receives the pointers of the input and output vectors, the size of the vectors, and word size in bytes. The operation is implemented using the RoCC *custom-2* instruction, and addition is defined by the operation code '1' in the *funct* field. Thus, using this interface and considering that no other RoCC extension accelerator coexist in the target system, a designer can define up to 32 distinct associative operations, that load up to two vectors and write to one, of arbitrary length and word size of up to 15 bytes.

Figure 5 shows the square matrix multiplication ($A \times B = C$) kernel using RV-Across. In this code, it is assumed that the linearized matrices A, B and C are already initialized in SPM using DMA. The constant *DIMENSION* represents the matrix dimension size. SET_RVA is an operation in the AP that works as a *memset*, copying an element for each position of the vector. The buffer used has the same size of *DIMENSION*. The application uses this buffer, that stores each element of array A, operates in parallel, per column, with all elements of array B, and accumulates the result in C. For each iteration of the internal loop, the result for a row of C is generated. The application gains performance through the parallelism provided by the AP, eliminating a third most internal loop used in the conventional matrix multiplication algorithm.

4.2. PIM operations

RV-across provides a library implementing associative logical and arithmetic operations using the algorithm explained in Section 3. Table 1 shows the associative operations implemented in our simulator and the number of execution passes, or AP execution cycles, they require according to the word length. The number of passes is defined as the number of comparisons needed in an Associative Processing operation. Besides comparisons, the operations may execute a variable number of writes, that depend on the result of the

Table 1. Number of comparison passes for associative operations implemented in RV-Across to a word length of n bits.

Associative operations	Number of passes
Unsigned multiplication	$4 \times n^2$
Addition, Subtraction	$4 \times n$
OR, XOR	$2 \times n$
NOT, AND, Shift Right and Left	n

comparisons. The sum of the number of passes and writes is the number of execution cycles needed to conclude the operation.

RV-Across comes with an extensible structure that allows the user to modify the existing operations or implement their own within its core. This structure has methods to configure the key, the mask, and the LUT, as well as performing comparisons and writing values to the data vector during associative processing. Then, to create new operations, the user just needs to describe the associative algorithm in terms of these base structures. Also, RV-Across supports a trace function to analyse the operation pass by pass, producing cycle-accurate information about the execution and aiding on the development and debugging of new associative algorithms.

4.3. Output

RV-Across generates a log for each step of the operation, showing the control registers and the SPM with the data of the operators. This data is provided for each operation both for didactic purposes and the user to control the simulation. Also, writing, comparison, match, and mismatch events are counted and recorded in a file so that the user can extract latency and energy metrics using an appropriate model of their simulation scenario.

5. Experimentation

We evaluate our proposal in two scenarios. In the first scenario, we compare our AP approach with a conventional single-core CPU (spike model). This demonstrates how our simulator represents associative processing operations, showing different behaviors between the performance of AP and CPU according to the input size in different applications. Then, in the second scenario, we show a performance comparison between the AP and a multicore CPU, both running a matrix multiplication kernel. This scenario demonstrates the associative processing efficiency to execute vectorized operations when compared with the overhead of including additional execution cores in the target system.

5.1. Single-core scenario

We choose three applications to execute in this comparison scenario: matrix multiply, checksum, and bitcount. The input sizes vary from the minimum possible value at step 1 to a packet size of 1500 bytes, for bitcount and checksum, or to a 100x100 bytes matrix for matrix multiply. The CPU model uses a naive serial implementation of the applications, optimized by the compiler, tracing memory accesses on the simulator. The matrix multiply algorithm parallelizes the computation of each row in the output matrix as shown in Figure 5. To compute the checksum, the data vector is divided into halves, and each half is summed in parallel, wordwise, repeating the operation in a divide-and-conquer approach. For bitcount, the number of active bits in all individual words in the data vector

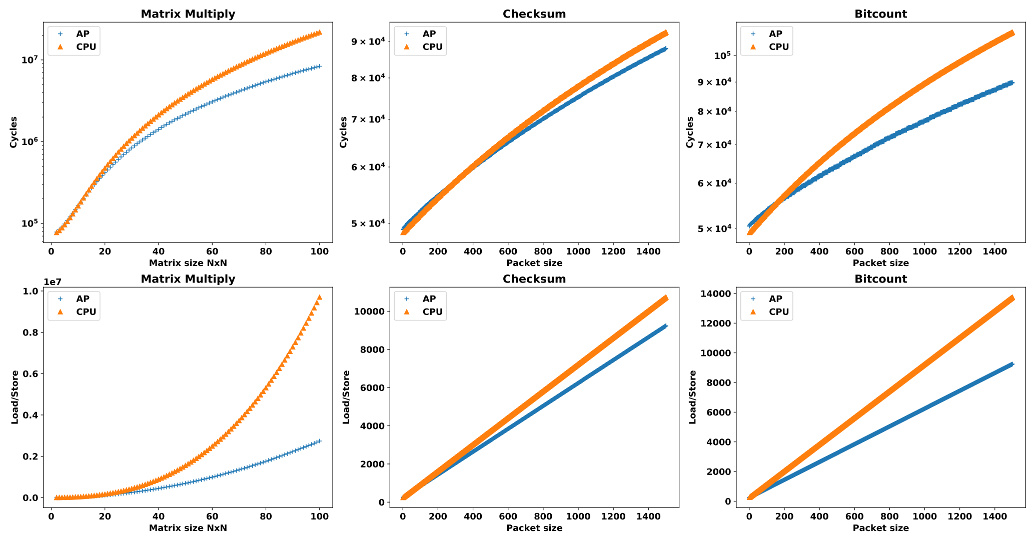


Figure 6. Number of cycles and memory accesses for AP and CPU execution.

is computed in parallel, and then the result is accumulated using the checksum algorithm. Our environment implements individual 32 KB L1 instruction and data caches and a single 128 KB L2 cache. The caches are only considered in memory accesses originated on the CPU. The data handled in the AP is transferred from the main DRAM using Direct Memory Access (DMA). In this scenario, the AP model uses at most 32 KB of the SPM for associative processing.

CPU instructions and memory accesses are the basis for calculating the performance of the CPU and AP models. To quantify each relevant operation of memory, AP, and CPU, we consider the latency of a single CPU instruction, L1 cache accesses, and AP individual comparison and writing to be 1 cycle. Accesses to the main memory and the L2 cache cost 100 and 10 cycles, respectively. The DMA latency is also assumed as 100 cycles. Since the SPM is a small memory block implemented close to the main processor, we assume that CPU and AP are clocked synchronously.

Figure 6 shows the number of cycles and load/store operations in the execution of each application, according to the input size. For small dimensions of data, the AP execution costs more than the CPU. The algorithm to execute the in-memory operation is more expensive than the dedicated CPU instructions to perform in a single data word because of the overhead imposed by the Associative Processor configuration. Nevertheless, differently from the CPU, the AP can execute the same operation in a wide data vector in parallel. As the data vector length grows, the AP execution increases in efficiency and overtakes the CPU execution. This intersection point occurs in the 11x11 bytes matrix size for matrix multiply, 182 bytes packet for checksum, and 152 bytes for bitcount. Even in the worst-case scenario (matrix size 2x2 bytes and packet size 1 byte), the AP overhead in comparison to the CPU execution is at most 2.5% for all applications.

For the maximum size of each input in the experiment using AP, matrix multiplication costs 39%, checksum 95%, and bitcount 82% of the CPU execution, leading to execution times 61%, 5% and 18% shorter, respectively. For all applications, less load/store operations are executed with associative processing, 71% in matrix multiplication, 13% in checksum, and 31% in bitcount due the avoidance of data movement between memory

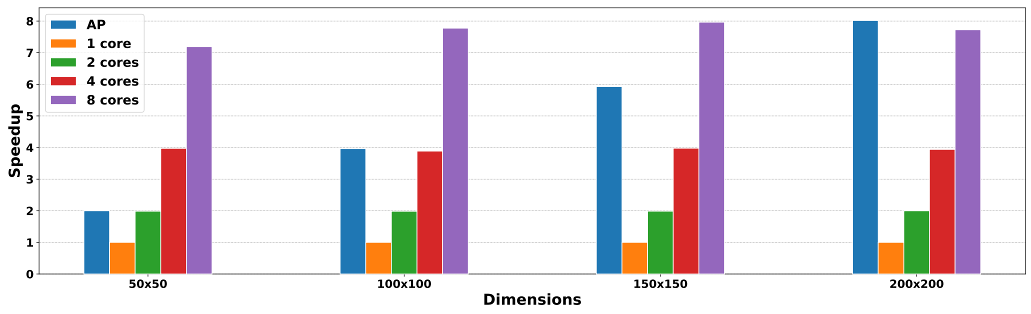


Figure 7. Associative Processing speed-up over a single-core CPU baseline, compared with speed-up for multi-threading with 2, 4, and 8 cores.

and CPU. The implementation overhead of the parallel operations is higher for checksum and bitcount than for matrix multiplication, and the data vector sizes are smaller, which explains the lower impact of PIM for these applications. However, the trends show that the larger are the data vectors, the higher is the performance increasing.

5.2. Multicore scenario

In the multicore scenario, we adapt the bare-metal multi-threaded matrix multiply reference implementation to use the AP modeling. Then, we compare the performance of both AP and CPU executing the multiplication kernel in 1, 2, 4 and 8 cores for matrices dimensions of 50x50, 100x100, 150x150, and 200x200 bytes. In AP, we consider that our SPM can store the input and output matrices and temporary values, which represents a size of at most 128 KB in the 200x200 scenario. The data is read from the main memory using DMA with a latency of 100 cycles. For the CPU, we disregard any influence of the memory hierarchy in the performance accountancy, assuming that all data is previously loaded in the lower latency level for computation, and thus the number of executed instructions determines the execution time. Thus, this evaluates the best-case scenario of the multicore execution in the comparison with the AP.

Figure 7 shows the speed-ups over of the AP and CPU with 2, 4, and 8 cores, using the single-core execution as baseline. In small data sizes (50x50), the control overhead of the AP dominates and affects performance, which makes its speed-up in the same order of magnitude of a 2-core CPU. Nonetheless, increasing the input size maximizes the performance gains that AP provides. From a 100x100 matrix, AP achieves a speed-up of 3.96x against 1.98x, 3.88x, and 7.77x of 2, 4, and 8 cores, respectively. This trending intensifies at a 200x200 input size, where AP overcome all tested multicore configuration, with 8.01x speed-up over 7.72x from the 8 core CPU. Furthermore, the performance gains of the AP approach scale up linearly with the dimension of the matrices, suggesting that a larger SPM can provide higher speed-ups for larger matrices even in comparison with larger multicore CPUs.

5.3. Simulation performance

Finally, we evaluate the performance of RV-Across itself to run the execution scenarios. Table 2 shows the average simulation time to run the matrix multiply application into the Vanilla Spike and the simulator modified with the RVA extensions, for a input 100x100 bytes matrices. Although RV-Across introduces significant overhead, the modified simulator includes routines to generate significantly more data to evaluate the execution, such

Table 2. Simulation Time for 100x100 matrix multiply.

	Vanilla simulator				RV-Across				
Simulations	1 core	2 cores	4 cores	8 cores	1 core	2 cores	4 cores	8 cores	AP
Avg time (s)	0.009	0.009	0.011	0.013	0.103	0.101	0.105	0.107	29.810

as instruction counters and memory accesses traces. Additionally, in the AP scenario, RV-Across emulates the associative operations step-by-step, generating statistics from algorithms, such as the number of passes, comparisons, matches, mismatches, writings, miswrites, and a full trace for the designer to understand the behavior of the operation. All these induces a lot of expensive I/O operations and impact significantly in execution time, but add data to get a more accurate simulation of the associative operations.

6. Conclusion

Associative processing is a PIM approach that has interested scientists and industry in the potential to process data in parallel and save energy by avoiding data movement. In this work, we present a simulator, RV-Across, which provides an interface for the user to test and validate associative operations in their applications as well as develop customized in-memory operations. Also, the simulator generates reports detailing all the steps of each operation and event of the Associative Processor. Thus, our tool exposes the behavior of an associative operation and the impact it has on the system, helping in the adoption of Associative Processing. RV-Across has the purpose of allowing the user to control and modify associative operations in the simulated system. We demonstrated the operation of RV-across by implementing and simulating three applications: matrix multiplication, checksum, and bitcount. Using a RISC-V infrastructure, we assess how much Associative Processing can affect a system. In the case of matrix multiplication with AP, 61% of execution cycles are saved in comparison to the application running on a single-core CPU, and overcomes multi-core CPUs providing speed-ups that increase linearly with matrices dimension. Thus, we demonstrate that PIM can be an interesting alternative for applications that move a lot of data, and the advantages increase the larger is the amount of data. Thus, our future work includes an energy assessment of the impact of AP in a system, understanding the trade-off between cost and size of an AP.

References

Boroumand, A., Ghose, S., Kim, Y., Ausavarungnirun, R., Shiu, E., Thakur, R., Kim, D., Kuusela, A., Knies, A., Ranganathan, P., and et al. (2018). Google workloads for consumer devices: Mitigating data movement bottlenecks. In *ASPLOS*, page 316–331.

Dai, G., Huang, T., Chi, Y., Zhao, J., Sun, G., Liu, Y., Wang, Y., Xie, Y., and Yang, H. (2019). GraphH: A processing-in-memory architecture for large-scale graph processing. *IEEE TCAD*, 38(4):640–653.

Gupta, S., Imani, M., Kaur, H., and Rosing, T. S. (2019). NNPIM: A Processing In-Memory Architecture for Neural Network Acceleration. *IEEE TC*, 9340(c):1–1.

Imani, M., Patil, S., and ŠimuniĆ Rosing, T. (2018). Approximate computing using multiple-access single-charge associative memory. *IEEE TETC*, 6(3):305–316.

Jeon, D. and Chung, K. (2017). CasHMC: A Cycle-Accurate Simulator for Hybrid Memory Cube. *IEEE CAL*, 16(1):10–13.

Kaplan, R., Yavits, L., Ginosar, R., and Weiser, U. (2017). A resistive cam processing-in-storage architecture for dna sequence alignment. *IEEE Micro*, 37(4):20–28.

Khoram, S., Zha, Y., and Li, J. (2018). An alternative analytical approach to associative processing. *IEEE CAL*, 17(2):113–116.

Kim, J. S., Senol Cali, D., Xin, H., Lee, D., Ghose, S., Alser, M., Hassan, H., Ergin, O., Alkan, C., and Mutlu, O. (2018). GRIM-Filter: Fast seed location filtering in DNA read mapping using processing-in-memory technologies. *BMC Genomics*, 19(2):89.

Lefurgy, C., Rajamani, K., Rawson, F., Felter, W., Kistler, M., and Keller, T. W. (2003). Energy management for commercial servers. *IEEE Computer*, 36(12):39–48.

Leidel, J. D. and Chen, Y. (2016). Hmc-sim-2.0: A simulation platform for exploring custom memory cube operations. In *IPDPSW*, pages 621–630.

Li, S., Liu, L., Peng Gu, Xu, C., and Yuan Xie (2016). NVSim-CAM: A circuit-level simulator for emerging nonvolatile memory based content-addressable memory. In *ICCAD*, pages 1–7.

Mutlu, O., Ghose, S., Gómez-Luna, J., and Ausavarungnirun, R. (2019). Enabling practical processing in and near memory for data-intensive computing. In *DAC*.

Nai, L., Hadidi, R., Sim, J., Kim, H., Kumar, P., and Kim, H. (2017). GraphPIM: Enabling instruction-level pim offloading in graph computing frameworks. In *HPCA*, pages 457–468.

Oliveira, G. F., Santos, P. C., Alves, M. A. Z., and Carro, L. (2017). A generic processing in memory cycle accurate simulator under hybrid memory cube architecture. In *SAMOS*, pages 54–61.

Paulo, J. and Lima, C. D. (2019). PIM-gem5 : a system simulator for Processing-in-Memory design space exploration. Master's thesis, Universidade Federal do Rio Grande do Sul.

Santos, P. C., de Lima, J. a. P. C., de Moura, R. F., Ahmed, H., Alves, M. A. Z., Beck, A. C. S., and Carro, L. (2018). Exploring IoT Platform with Technologically Agnostic Processing-in-memory Framework. In *INTESA*, pages 1–6.

Xu, S., Chen, X., Wang, Y., Han, Y., Qian, X., and Li, X. (2019). PIMSim: A flexible and detailed processing-in-memory simulator. *IEEE CAL*, 18(1):6–9.

Yang, X., Hou, Y., and He, H. (2019). A processing-in-memory architecture programming paradigm for wireless internet-of-things applications. *Sensors*, 19:140.

Yantir, H. E., Eltawil, A. M., and Kurdahi, F. J. (2018). A hybrid approximate computing approach for associative in-memory processors. *IEEE JETCAS*, pages 1–1.

Yavits, L., Kaplan, R., and Ginosar, R. (2018). PRINS: resistive CAM processing in storage. *CoRR*, abs/1805.09612.

Yavits, L., Morad, A., and Ginosar, R. (2015). Computer architecture with associative processor replacing last-level cache and simd accelerator. *IEEE TC*, 64(2):368–381.

Zhang, D., Jayasena, N., Lyashevsky, A., Greathouse, J. L., Xu, L., and Ignatowski, M. (2014). TOP-PIM: Throughput-oriented programmable processing in memory. In *HPDC*, pages 85–98.