Universidade Estadual de Campinas
Instituto de Computação

Jonathas Evangelista da Silveira

Exploring Associative Processing with the RV-Across simulator

Uma exploração de Processamento Associativo com o simulador RV-Across

CAMPINAS
2021

Jonathas Evangelista da Silveira

Exploring Associative Processing with the RV-Across simulator

Uma exploração de Processamento Associativo com o simulador RV-Across

Dissertação apresentada ao Instituto de Computação da Universidade Estadual de Campinas como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação.

Dissertation presented to the Institute of Computing of the University of Campinas in partial fulfillment of the requirements for the degree of Master in Computer Science.

Supervisor/Orientador: Prof. Dr. Lucas Francisco Wanner

Este exemplar corresponde à versão final da Dissertação defendida por Jonathas Evangelista da Silveira e orientada pelo Prof. Dr. Lucas Francisco Wanner.

CAMPINAS
2021

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca do Instituto de Matemática, Estatística e Computação Científica
Ana Regina Machado - CRB 8/5467

Si39e

Silveira, Jonathas Evangelista da, 1996-
Exploring associative processing with the RV-Across simulator / Jonathas Evangelista da Silveira. – Campinas, SP : [s.n.], 2021.

Orientador: Lucas Francisco Wanner.
Dissertação (mestrado) – Universidade Estadual de Campinas, Instituto de Computação.

1. Sistemas de memória de computadores. 2. Processamento associativo. 3. Eficiência energética. I. Wanner, Lucas Francisco, 1981-. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Informações para Biblioteca Digital

Título em outro idioma: Uma exploração de processamento associativo com o simulador RV-Across
Palavras-chave em inglês:
Computer storage devices
Associative processing
Energy efficiency
Área de concentração: Ciência da Computação
Titulação: Mestre em Ciência da Computação
Banca examinadora:
Lucas Francisco Wanner [Orientador]
Alba Cristina Magalhães Alves de Melo
Rodolfo Jardim de Azevedo
Data de defesa: 12-03-2021
Programa de Pós-Graduação: Ciência da Computação

Identificação e informações acadêmicas do(a) aluno(a)
- ORCID do autor: https://orcid.org/0000-0002-7398-3947
- Currículo Lattes do autor: http://lattes.cnpq.br/2403081259708319

Universidade Estadual de Campinas
Instituto de Computação

Jonathas Evangelista da Silveira

Exploring Associative Processing with the RV-Across simulator

Uma exploração de Processamento Associativo com o simulador RV-Across

Banca Examinadora:

- Prof. Dr. Lucas Francisco Wanner
  IC/UNICAMP
- Profa. Dra. Alba Cristina Magalhães Alves de Melo
  CIC/Universidade de Brasília
- Prof. Dr. Rodolfo Jardim de Azevedo
  IC/UNICAMP

A ata da defesa, assinada pelos membros da Comissão Examinadora, consta no SIGA/Sistema de Fluxo de Dissertação/Tese e na Secretaria do Programa da Unidade.

Campinas, 12 de março de 2021

Acknowledgements

First, I thank God for providing me all the knowledge I have (*Prv* 1 : 7) and for the salvation (*Eph* 2 : 8) for the grace. As Abraham Kuyper (ex-minister of the Netherlands) once said, and I agree: "There is not a square inch (including the *bits*) in the whole domain of our human existence over which Christ, who is sovereign over all, does not say: 'Mine!'".

Second, I thank my family for providing me with all the necessary spiritual, emotional and financial support. I am grateful to my wife Camila Silveira who was with me throughout my master's degree holding me and acting graciously towards me. I thank my parents Denilson Silveira and Claudiane Silveira for being models of effort, work and dedication. Something that science cannot explain. I am also grateful for the support of my brother Alexander Silveira and his wife Edna, who have always been around in difficult times. I am grateful for the unconditional support of my father-in-law João Filho, my mother-in-law Dona Leonice Abreu and my brother-in-law João Paulo. Finally, I thank two great men, Laysson Luz and Ramon Nepomuceno, who helped me a lot and that I consider as my family.

Third, I thank my advisor prof. dr. Lucas Wanner for giving me the opportunity to learn more about computing and for guiding me through these two years of learning. I also thank my undergraduate advisor prof. dr. Ivan Silva for teaching about computing and life. I thank my companions Isaías Felzmman and João Filho who helped me in my work, especially in scientific production. Finally, I thank my colleagues at LSC who taught me a lot about science and made this period lighter.

Finally, I thank Unicamp for all its support as an institution. This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001.

Resumo

Múltiplos trabalhos apontam para um gargalo de desempenho entre o processador e a memória. Esse gargalo se destaca na execução de aplicações, como o Aprendizado de Máquina, que processam uma grande quantidade de dados. Nessas aplicações, a movimentação de dados representa uma parcela significativa tanto em termos de tempo de processamento quanto de consumo de energia. O uso de novas arquiteturas multi-core, aceleradores e Unidades de Processamento Gráfico (*Graphics Processing Unit* — GPU) pode melhorar o desempenho desses aplicativos por meio do processamento paralelo. No entanto, a utilização dessas arquiteturas não elimina a necessidade de mover dados, que passam por diferentes níveis de uma hierarquia de memória para serem processados.

Este trabalho explora o Processamento em Memória (*Processing in Memory* — PIM), especificamente o Processamento Associativo, como alternativa para acelerar as aplicações, processando seus dados em paralelo na memória permitindo melhor desempenho do sistema e economia de energia. O Processamento Associativo fornece computação paralela de alto desempenho e com baixo consumo de energia usando uma Memória endereçável por conteúdo (*Content-Adressable Memory* — CAM). Através do poder de comparação e escrita em paralelo da CAM, complementado por registradores especiais de controle e tabelas de consulta (*Lookup Tables*), é possível realizar operações entre vetores de dados utilizando um número pequeno e constante de ciclos por operação.

No trabalho, analisamos o potencial do Processamento Associativo em termos de tempo de execução e consumo de energia em diferentes *kernels* de aplicações. Para isso, desenvolvemos o RV-Across, um simulador de Processamento Associativo baseado em RISC-V para teste, validação e modelagem de operações associativas. O simulador facilita o projeto de arquiteturas de processamento associativo e próximo à memória, oferecendo interfaces tanto para a construção de novas operações quanto para experimentação de alto nível. Criamos um modelo de arquitetura para o simulador com processamento associativo e o comparamos este modelo com as alternativas baseadas em CPU e *multi-core*. Para avaliação de desempenho, construímos um modelo de latência e energia fundamentado em dados da literatura. Aplicamos o modelo para comparar diferentes cenários, alterando características das entradas e o tamanho do Processador Associativo nas aplicações.

Nossos resultados destacam a relação direta entre o tamanho dos dados e a melhoria potencial de desempenho do processamento associativo. Para a convolução 2D, o modelo de Processamento Associativo obteve um ganho relativo de 2x em latência, 2x em consumo de energia, e 13x no número de operações de *load/store*. Na multiplicação de matrizes, a aceleração aumenta linearmente com a dimensão das matrizes, atingindo 8x para matrizes de 200x200 bytes e superando a execução paralela em uma CPU de 8 núcleos. As vantagens do Processamento associativo evidenciadas nos resultados revelam uma alternativa para sistemas que necessitam manter um equilíbrio

entre processamento e gasto energético, como os dispositivos embarcados. Finalmente, o ambiente de simulação e avaliação que construímos pode habilitar mais exploração dessa alternativa em diferentes aplicações e cenários de uso.

Abstract

Many works have pointed to a performance bottleneck between Processor and Memory. This bottleneck stands out when running applications, such as Machine Learning, which process large quantities of data. For these applications, the movement of data represents a significant fraction of processing time and energy consumption. The use of new multi-core architectures, accelerators and Graphic Processing Units (GPU) can improve the performance of these applications through parallel processing. However, utilizing these architectures does not eliminate the need to move data, which are transported through different levels of a memory hierarchy to be processed.

Our work explores Processing in Memory (PIM), and in particular Associative Processing, as an alternative to accelerate applications, by processing data in parallel in memory, thus allowing for better system performance and energy savings. Associative Processing provides high-performance and energy-efficient parallel computation using a Content-Addressable Memory (CAM). CAM provides parallel comparison and writing, and by augmenting a CAM with special control registers and Lookup Tables, it is possible to perform computation between vectors of data with a small and constant number of cycles per operation.

In this work, we analyze the potential of Associative Processing in terms of execution time and energy consumption in different application kernels. To achieve this goal we developed RV-Across, an Associative Processing Simulator based on RISC-V for testing, validation, and modeling associative operations. The simulator eases the design of associative and near-memory processing architectures by offering interfaces to both building new operations and performing high-level experimentation. We created an architectural model for the simulator with associative processing and evaluated it by comparing it with the CPU-only and multi-core models. The simulator includes latency and energy models based on data from literature to allow for evaluation and comparison. We apply the models to compare different scenarios, changing the input and size of the Associative Processor in the applications.

Our results highlight the direct relation between data length and potential performance and energy improvement of associative processing. For 2D convolution, the Associative Processing model obtained a relative gain of 2x in latency, 2x in energy, and 13x in the number of load/store operations. For matrix multiplication, the speed-up increases linearly with input dimensions, achieving 8x for 200x200 bytes matrices and outperforming parallel execution in an 8-core CPU. The advantages of associative processing shown in the results are indicative of a real alternative for systems that need to maintain a balance between processing and energy expenditure, such as embedded devices. Finally, the simulation and evaluation environment we have built can enable further exploration of this alternative for different usage scenarios and applications.

List of Figures

3.1 Associative Processor overview. 20
3.2 Associative processing diagram. 20
3.3 Associative XOR operation. 23
3.4 LUTs for AND, OR and NOT operations. 23
3.5 Associative ADD operation. 24
3.6 Cycle by cycle simulation on the WebAP. 25

5.2 RV-Across architectural model. 33
5.1 RV-Across design flow. 33
5.3 RoCC instructions library. 34
5.4 Output of RV-Across with the last cycle of the associative addition operation and the report of that operation. 36
5.5 UML showing the APrv inheriting attributes and methods from APTemplate for the RV-Across simulation. 40

6.1 Associative algorithm for matrix multiply using custom instructions for implementing row parallelism. 41
6.2 Overview of the behavior of matrix multiply in associative processing. . 42
6.3 Matrix extracted from the first element of the image operating with part of the kernel. 43
6.4 Overview of the behavior of 2D convolution in associative processing. . 44
6.5 Associative algorithm for 2D convolution using custom instructions for implementing row parallelism. 44
6.6 Associative algorithm for ReLU using special instruction. 45
6.7 Simulation of the ReLU operation within the CAM. 46

7.1 Latency, energy consumption and memory accesses for AP and CPU, both performing bitcount. 48
7.2 Latency, energy consumption and memory accesses for AP and CPU, both performing checksum. 48
7.3 Latency, energy consumption and memory accesses for AP and CPU, both performing 1D convolution. 49
7.4 Latency, energy consumption and memory accesses for AP and CPU, both performing hamming distance calculation. 49
7.5 Latency, energy consumption and memory accesses for AP and CPU, both executing Manhattan distance. 50
7.6 Latency, energy consumption and memory accesses for AP and CPU, both executing ReLU. 50
7.7 Latency, energy consumption and memory accesses for AP and CPU, both performing matrix multiply. 51

7.8 Latency, energy consumption and memory accesses for AP and CPU. The CPU performing naive 2D convolution algorithm. 51
7.9 Latency, energy consumption and memory accesses for AP and CPU. The CPU performing optimized 2D convolution algorithm. 52
7.10 Comparison in number of cycles between the normal and optimized version of the 2D convolution in the AP, separating the executions in phases for initialization, data organization and computation in the AP. . 52
7.11 Latency, energy consumption and memory accesses for AP and CPU. The AP and CPU, both performing optimized 2D convolution algorithm. 53
7.12 Energy consumption for different AP sizes, comparing the normal and optimized for AP and optimized version for CPU. 53
7.13 Energy consumption for different AP sizes, comparing AP and CPU, performing matrix multiply. 54
7.14 AP performing matrix multiply with different word sizes. 54
7.15 The relative improvement in terms of latency, energy consumption and load/stores for all applications evaluated. 55
7.16 Associative Processing matrix multiplication speed-up over a single-core CPU baseline, compared with speed-up for multi-threading with 2, 4, and 8 cores. 55

Contents

1 Introduction **13**

2 Background **15**
2.1 RISC-V 15
2.2 Processing in Memory 16
2.3 3D Memory 16
2.4 Alternative Solutions 17
2.5 Associative Processing 18

3 Associative Processing **19**
3.1 Associative Processor 19
3.2 Constructing operations 22
3.3 Logic operations 22
3.4 Arithmetic operations 24
3.5 Web AP simulator 25

4 Related Work **26**
4.1 Processing in Memory 26
4.2 3D Memories 27
4.3 Associative Processors 28
4.4 Simulators 28
4.5 Comparison 29

5 RV-Across: An Associative Processing Simulator **32**
5.1 Overview 32
5.2 User interface 34
5.3 Output 35
5.4 Latency model 36
5.5 Energy model 38
5.6 Customizing associative operations 39

6 Case studies **41**
6.1 Matrix Multiplication 41
6.2 2D Convolution 43
6.3 ReLU 45

7 Demonstration and Results **47**
7.1 AP vs CPU 47
7.2 AP vs Multicore 54

7.3 Simulation performance 56

8 Conclusion **57**

Chapter 1

Introduction

Researchers have sought to develop solutions in hardware that achieve high performance in terms of execution time, for specific or general purposes, while at the same time optimizing energy consumption. Many of these solutions are based on *Von Neumann* architectures, and therefore, rely on processing separated from memory. Since memory is utilized just for storage, multiprocessors, accelerators, and GPUs need to transport the data from memory for computation, spending a lot of time and energy in data movement.

Emerging data-intensive workloads, such as *Neural Network*, *Image Processing*, *Graphs* and *DNA Alignment* require fast data transfers for efficient computation. Because data transfer rates are typically much lower than maximum processing rates, CPUs often must wait for data, limiting system throughput. This difference in data transfer rates and processing speeds, known as the Von Neumann bottleneck [35], has been increasing with subsequent hardware generations, leading to the so-called *memory wall* [37, 42]. In addition to potentially limiting performance, the movement of data consumes a significant amount of the overall energy of the system. Data movement between the core and off-chip memory incurs ~100x higher energy than a floating-point operation [34, 39]. In Google Docs scrolling, for example, moving data represents more than 30% of energy consumption [6], and memory can account for up to 41% of the energy consumption of an entire server system [30].

Given this scenario, *Processing in Memory* (PIM) is an alternative for saving energy by avoiding data transfers and getting better performance using parallel computation. With the breakdown of Dennard Scaling and the eminent end of Moore's Law, academy and industry have shown increased interest in this alternative [35]. 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 [12, 18, 29, 36].

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 [20, 26, 51]. However, the lack of a flexible simulation infrastructure for the test and validation of in-memory operations is still an obstacle to enable its adoption [35].

In this work we show how Associative Processing works and how to build an algorithm in this approach and its performance, assessing speed and energy. To achieve these contributions, we developed the RV-Across (RISC-V Associative Processing Simulator). RV-Across is a simulator developed with the aim of providing easy programming, test, and evaluation of Associative Processing or near-memory processing context. Thus, taking its advantages, we implement applications in different scenarios and compare with a CPU approach to observe the potential of Associative Processing as well as its trade-off. The main contributions we present in this work are:

- Exploration and detailing of Associative Processing Algorithms for matrix multiplication, 2D convolution and ReLU function;
- An extensible simulation tool that enhances associative processing evaluation;
- An architectural model to interface PIM operations with an Associative Processor;
- A simple programming model for Associative Processing;
- Case studies using Associative Processing operations in application kernels;
- A performance analysis of applications in the Associative Processing model.

Our experiments highlight the potential of Associative processing and how the simulator works. In a model using Associative Processing, in most cases, resulted in fewer accesses to memory and consequently higher performance. For matrix multiply, associative processing obtained a relative improvement of 6x to latency, 4x to energy and 28x to load/store operations. For 2D convolution, in the best scenario 2x, 2x and 13x to latency, energy consumption and load/stores respectively. When considering a multi-core 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. We evaluate the Associative Processor in various sizes to balance area and energy consumption. Processor with 16 KB or 32 KB is ideal, occupying an area smaller than a RISC-V core, proving to be a viable alternative for embedded systems.

This work is structured in eight chapters from introduction.The Chapter 2 provides key concepts to understand the work context. The Chapter 3 contains the implementation and simulation of a logic/arithmetic Associative Processing operation. Chapter 4 describes and classify related works, indicating similarities and differences among them and our work. Chapter 5 details the RV-Across from overview until the output of simulator. The Chapter 6 presents cases of applications adapted for associative processing using RV-Across. Chapter 7 shows the evaluation modes and performance results of Associative Processing in different scenarios. Lastly, Chapter 8 displays our conclusions and future work.

Chapter 2

Background

This Chapter explains in detail the fundamentals to understand our work and its context. The first section explains what is RISC-V, and its benefits. After, it is described the concept of Processing in Memory and a summary of the main approaches as well as works related to each branch. The third section talks about 3D memories and works that utilize them to get better performance. The fourth section shows works about PIM which are out of 3D memory and Associative scope. Finally, we explore how an Associative Processor is designed and how it works.

2.1 RISC-V

RISC-V is an *Instruction Set Architecture* (ISA) which follows the *Reduced Instruction Set Computing* (RISC) pattern and was designed with the didactic and scientific goals. The name RISC-V was chosen to represent the fifth RISC ISA developed by UC Berkley. RISC-I, RISC-II, SOAR and SPUR, in sequence, were the other versions [2].

This ISA has been adopted due to its benefits. It is an ISA open-source for industry and academy. There are free RISC-V cores with *Field Programmable Gate Array* (FPGA) support. RISC-V has float-point support and can be 32 or 64 bits. Rocket Chip, a RISC-V system on chip (SoC), for example, can be easily extended to a multi-core. Also, it is a platform where it is possible to add modules and instructions, creating variations. Also, RISC-V has a powerful tool-chain contained simulators, compilers and open documentation. These resources led to industry to fabricate chips using RISC-V.

The RISC-V obtains an interface for binding accelerator and additional modules. In the case of Rocket Chip, the custom instructions fulfill this role. The most used RISC-V models that can be extended are BOOM and Rocket Chip. In view of the present work, we utilized RISC-V and custom instructions for communication with the Associative Processing module on RV-Across. Chapter 5 explains in details about that programming interface and how to use it.

2.2 Processing in Memory

Processing In Memory (PIM) is a model that increases hardware performance in terms of bandwidth (about memory and CPU), latency, and energy consumption. It performs the computation in memory, where the application data is located. The PIM concept goes in the opposite direction to processing model adopted currently, the *Von Neummann model*. PIM is an approach applied in different ways, all looking for the same goal, of processing without data movement.

Initially, in the 70's, STARAM was developed, which is based on PIM model. This architecture was built by Goodyear and had as the purpose to change the processing manner. STARAM is constructed for a bunch of Associative Processors (APs). Recently, researches are joining associative processing with new memory trends as *Non Volatile Memories* (NVMs) and approximate computing [51, 54]. ReCAM, for example, is an implementation of a CAM using resistive memory.

Following the same fashion of associative computing, *Content-Addressable Memory* (CAM), other PIM production, is applied in routers, specifically on network layer for high performance *Internet Protocol* (IP) comparisons. The CAM indexes its contents using other content, as in a hash table. Thus, a router, after extracting the packet's IP address, inserts this address in the CAM, and it quickly returns (1 cycle) the destination address.

In addition to AP and CAM, recently academy and industry try to implement Processing in Memory in both, DRAM and 3D memories [10]. These components execute logic and arithmetic operations in their storage without undergoing major architectural modifications [35]. However, there are barriers that all approaches need to overcome: Making programming possible and easy for PIM systems, providing data coherency among PIM module and other system's modules, and building a simulation infrastructure for analogical and digital simulation. Our work has as goals to afford knowledge about associative processing and to provide a tool that is able to give a programming interface and application for validation of the potential of PIM. The same has been done in the context of 3D memories. There are many works of simulators and interfaces for programming and testing the resources provided by 3D logic.

2.3 3D Memory

3D memory is a new technology that stacks vertically the logic layer of the memory in three dimensions. That new memory makes better use of die space with storage stacked. *Through Silicon Vias* (TSV), which are the interconnections within a die stack, enable high bandwidth, lower latency, and energy savings on communications among dies. 3D memory is able to access data in parallel due to three-dimensional architecture and components as TSV.

The 3D memory structure creates a perfect environment for Processing in Memory. For that reason, some 3D memories already support logical and arithmetic operations in parallel. An adverse effect of stacking in-memory 3D is increased peak temperature.

Also, volatile and non-volatile 3D memories have been produced by industry on large scale, aiming for high-performance.

Researchers have used the 3D memory capabilities in different scenarios and applications [16, 24, 27, 44]. For example, the architecture of Ahn et. al [1] and Zhang et. al. [59] utilized *Hybrid Memory Cube* (HCM), a 3D memory with PIM support, to accelerate data-intensive applications. Besides developing architecture, both created an interface (programming model) and treated coherency problems. Compared to *Graphics Processing Unit* (GPU), both achieved better performance in execution time and energy consumption. Dai et. al. [12], Zhang et. al. [60] and Nai et. al. [36] used the 3D memory potential to process graphs in parallel. Similarly, Kim et. al. [29] applied PIM with 3D memory in the DNA sequence alignment. They developed a simulation platform for test, validation, and evaluation of systems with 3D memories, providing a programming interface to the user.

The RV-Across, our simulation platform, has similarities with PIM-gem5, work produced by Santos et. al. [44], by given a flexible way of simulation. But, instead of 3D memory, our simulator has Associative Processing features. In general, our work, as 3D memories works, uses PIM concept, but focused in a specific technique.

2.4 Alternative Solutions

Some works used PIM, but do not even fit in 3D memory or Associative Processor. Chi et al. [8] proposed a PIM architecture, called PRIME, to accelerate *Neural Network* (NN) applications in *Resistive random-access memory* (ReRAM) based main memory. In PRIME, they insert *full functions* accelerators into memory that executes NN operations like search and convolutions. They designed complete software/hardware full functions for developers.

Gupta et al. [18] created an architecture, NNPIM, with full support, in terms of operations, to execute the inference phase inside the memory. They design a crossbar memory architecture that supports fast addition, multiplication, and search operations inside the memory. For search, they took advantage of the inherent characteristics of capacitors. Shafiee et al. [46] explored processing in approach, where memristor crossbar arrays not only store input weights but also perform dot-product operations in an analog manner.

Imani et al. [19] constructed a PIM accelerator, GenPIM, in an NVM which supports bitwise operations, search operation, addition, and multiplication. In GenPIM, conventional cores are connected to PIM accelerators. The GenPIM replaces DRAM with non-volatile memory since NVM can support both memory and processing functionalities. Seshadri et. al. [45] proposed Ambit, an accelerator in memory for bulk bitwise operations. Ambit exploits the analog operation of DRAM technology to execute bitwise operations inside DRAM, thereby exploiting the full internal DRAM bandwidth. Ambit consists of two features: simultaneous activation of three DRAM rows that share the same set of sense amplifiers, enabling the system to perform bitwise AND and OR

operations, and the ability to use the inverters present inside the sense amplifier to perform bit-wise NOT operations.

Our work has the same intention and motivation as those of the aforementioned researches, however, using associative processing to accelerate applications. Associative processing is a way of doing PIM already applied and implemented, with well-defined behavior, being utilized in various memory technologies.

2.5 Associative Processing

Associative processing is a way of parallel processing in an associative memory structure. Utilizing sequences of comparisons and writings based on Lookup Table, which represents the truth table of the operation, the AP executes arithmetic and logical operations between vectors in parallel spending a constant number of cycles [52, 54]. To enable associative processing, the AP takes advantage of a completely associative memory, the CAM. By adding a small overhead of special registers, the states of the implemented operations and the controller containing the algorithms, the AP acquires the ability to perform operations in parallel. We explained the structure, the algorithm and their behavior in the Chapter 3.

Nowadays scientists are utilizing associative processing in NVMs, in order to get a smaller cell area, a lower leakage power, and an increased overall chip area efficiency [3, 52, 55]. Also, associative processing, in academy, is applyed to accelerate various applications as DNA alignment, Database operations and CNN [32, 51, 57]. We, in this work, implemented application kernels and evaluated energy and time, simulating, on RV-Across, CPU model versus the AP model, in order to get a vision about Associative Processor potential.

Chapter 3

Associative Processing

This Chapter explains the structure used to perform associative processing and how to build operations on it with or without optimizations. The entire structure is detailed as well as its function in the execution of a generic operation, which is illustrated both in the diagram and in pseudo-code. Moreover, it shows step by step how to construct an operation from scratch. Also, the Chapter illustrates simple simulations of logic and arithmetic associative operations, containing the Associative Processor state for each cycle.

3.1 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 [52, 54]. 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:

- *Controller*: Module which is responsible for instruction decoding and configuration of other components for associative operation.
- *CAM*: Associative Memory in which the data will be stored. A CAM is able to recover data from its content.
- *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.

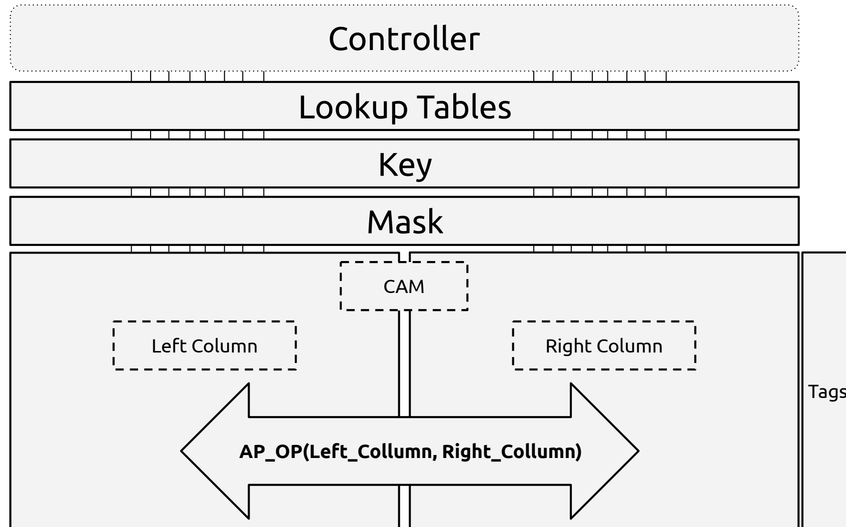

Figure 3.1: Associative Processor overview.

Figure 3.1 illustrates the Associative Processor overview with its components. As shown in Figure 3.1, the CAM is divided into data columns where one column can reference another. The CAM executes a search in one cycle by the capability to manipulate every bit (comparison and writing) of its storage. The CAM search is powerful, but it generates high area overhead. The AP takes advantage of CAM functionalities to perform operations. It uses those columns as vectors, and through the logic of comparison and writings based on LUT, which works like a truth table, does parallel operations between vectors in a constant number of cycles (AP_OP in Figure 3.1).

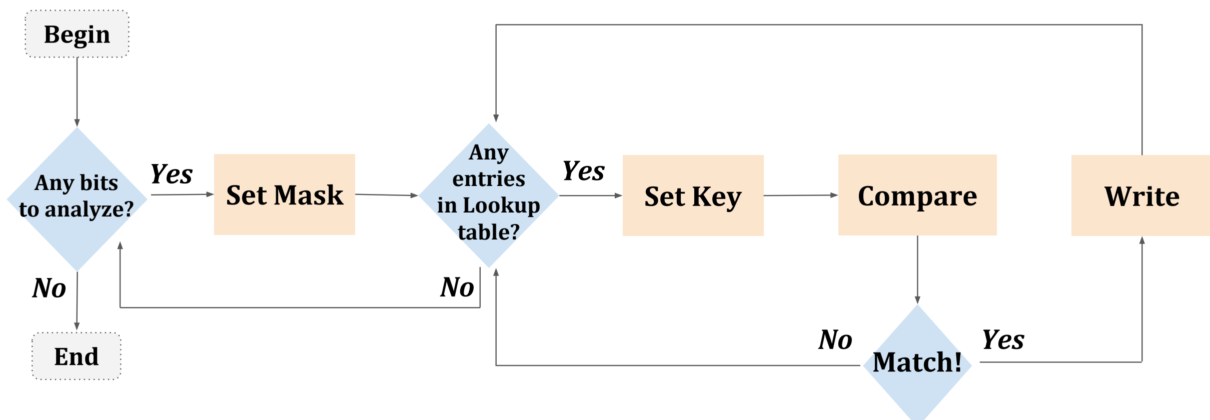

Figure 3.2: Associative processing diagram.

Associative processing can be summarized in three phases: selection, comparison, and writing of bits. First, the *Mask* register is set to select the columns (operands) 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 3.2 illustrates a diagram showing this operation flow.

Algorithm 1 Generic associative processing algorithm (R[] = A[] OP B[])

```
1: for i = 0, 1, ...,wordSizeA do
2:     for j = 0, 1, ...,wordSizeB do
3:         setMask(R, i, j)
4:         for pass = 1, 2, ..., sizeLUT do
5:             setKey(LUT[pass], R, i, j)
6:             Compare(Key, Mask, A[i], B[j])
7:             Write(LUT[pass], R)
8:         end for
9:     end for
10: end for
```

Algorithm 1 is the implementation of a generic associative processing algorithm performing an operation between two vectors (R[] = A[] OP B[]). Notice that the two external loops with the mask setting represent the selection phase (line 2). In other words, the bit selection, of vectors A and B, will be operated. After selection, for each pass from LUT, in an interval of two cycles, the key is configured, compared to bits selected, and the writing is triggered if **Match** happens. So, finalizing the comparison and writing phase. The algorithm ends when all passes are compared and/or written on all bits.

The algorithm shown represents a common shape of an associative algorithm. It is important to emphasize that associative processing does not have a unique algorithm format. The way the associative algorithm is designed changes according to the operation. Both the mask and the key registers can be configured via the controller, depending on the associative operation. The AP is flexible for building the LUTs and modifying the registers, however, there is a rigid data organization format for correct execution.

This section was based on works produced by Yantir et al. [51]. Our simulator, RV-Across, implements the associative addition and multiplication described in their work, as well as the cycle model. We used the same mechanism as Yantir at al. to build other LUTs, and thus create new operations (subtraction, AND, OR, XOR, and NOT).

3.2 Constructing operations

Associative operations are powerful due to the capability of applying the same function into all data in parallel without data movement, just using comparisons and writings. For the construction of the operation, the developer needs, initially, to understand the behavior of its operation to set mask and key registers correctly. Another step is to define inputs and outputs as a truth table. The truth table will be the LUT. The input values represent the comparison bits and output values the bits that will be writing.

There are two optimizations for associative operation [52]. The first decrease the number of passes from LUT. The developer could exclude entries of the LUT by initializing the output vector with zeros or ones, depending on which is the majority among the outputs. For example, if the output vector starts with zeros, the developer would exclude the three passes with output '0' of AND operation. It is as if the vector already had the pre-written results. The second optimization saves storage by doing destructive operations. Instead of, for example, $C[] = A[] + B[]$, the performed operation would be $A[] = A[] + B[]$. That technique saves a third of storage space per individual operation. For example, a non-destructive add operation, with one byte of word size, between two vectors containing 4 elements each, spends 12 bytes of the AP. Using the optimization, the operation would spend just 8 bytes.

Applying or not the optimizations, it is important to verify the order of LUT passes, because it influences directly over the result. One pass can interfere with another pass by writing at the wrong moment. For a correct associative operation, it is fundamental to test a bunch of samples, following and verifying cycle by cycle the results. In the next sections, associative logic and arithmetic operation are detailed through simulations.

3.3 Logic operations

Associative logic operations are simple to design and implement. It is just to use the standard truth table as LUT, and apply overall bits. Figure 3.3 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.

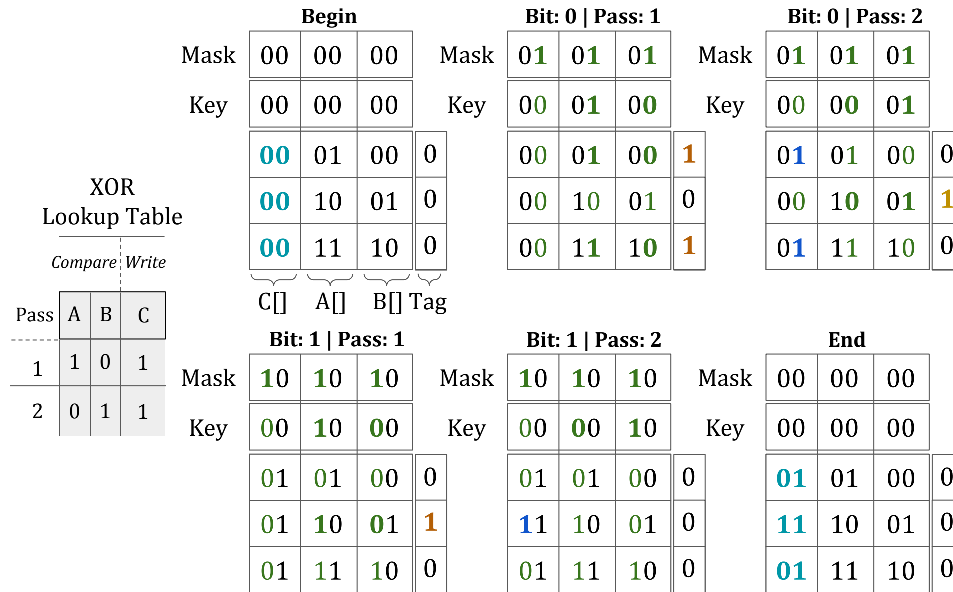

Figure 3.3: Associative XOR operation.

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 resulting associative XOR. Thus, the operation can be performed in constant time for N elements of a vector without moving data.

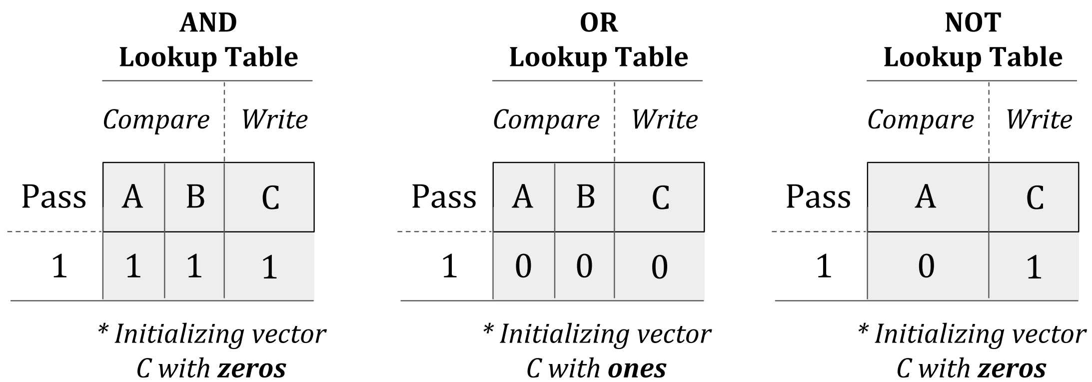

Figure 3.4: LUTs for AND, OR and NOT operations.

Figure 3.4 presents optimized LUT for AND, OR and NOT. With those tables and adopting the same shape of the mask of the XOR operations simulated, it is possible to perform each one. For the arithmetic operations, this shape can change depending on the operation. The next section explains the difference between arithmetic and logic associative operations.

3.4 Arithmetic operations

Unlike logic associative operations, arithmetic operations require more complexity to be implemented. In addition to considering the operands and results for comparison and writing, it is necessary to deal with the carry. The carry influences directly on the optimization that reduces LUT passes, because its columns can not be removed even with pre-written results. In general, the carry is written in the most significant bit, far from the main computation that occurs in other bits.

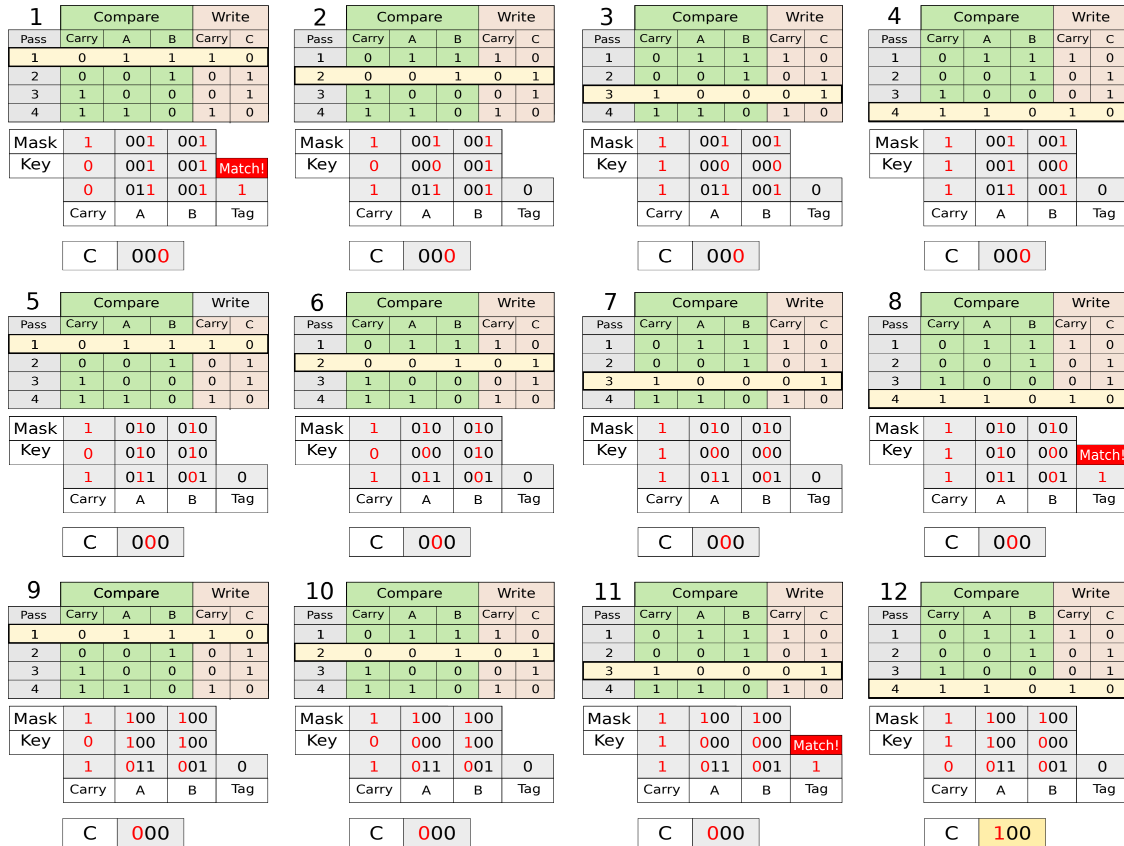

Figure 3.5: Associative ADD operation.

Figure 3.5 shows a simulation of an associative ADD operation between ($C[0] = A[3] + B[1]$). As in the XOR example, the ADD works in parallel. But in this case, to facilitate the understanding, the simulation was reduced for execution between two operands. The Figure is divided into 12 subfigures with the state for each pass. In the subfigure, the LUT, the full adder truth table summarized (operation states), is above the table of operands. The table of operands contains the state of special registers and the operands. For each bit, all passes from the LUT are applied for comparisons, therefore, in the case of addition which has 4 passes, will be $4 \times n$ comparisons to finish the operation (n number of bits). In this example, the ADD operation has 12 comparisons.

In this operation, the AP selects the bits, through mask register, like XOR operation. Starting from LSB to MSB, the AP does comparisons and writings based on the LUT. Notice that the selection of the mask is represented in the table of operands by the bits highlighted in red. Also, the comparison bits from LUT (green area) are represented for

the Key aligned with the Mask bits. When the Key bits aligned with Mask are equals to operands bits (A, B and Carry), occurs a Match and the Tag is set to 1. Hence, the writing bits from LUT (pink area) referring to the current pass are written in both carry and C. We can see a Match in the first subfigure, the Key bits equal to operands bits and the Tag receiving 1 to signalize the writing in the next cycle. In the next subfigure, observe that the writing bits for the pass 1, which are 1 to Carry and 0 to C, were applied.

The AP performs massive parallel selections, comparisons, and writings through the controller and special registers. This means that AP spends a constant latency (oriented by the word size of the CAM) to execute an operation regardless of the amount of data. Also, the AP parallelism provides high throughput in addition to saving energy. This kind of computation into the memory serves to accelerate applications that perform the same procedure in a set of data several times. In this work, we need to simulate an AP, with all logic and arithmetic operations, in an architectural model, aiming to compare with CPU alone and multi CPU approaches. For this purpose, we built the RV-Across which is the Spike, a standard RISC-V ISA simulator and adding an extension that contains the AP behavior. In the next Chapter, the RV-Across is explained in detail.

3.5 Web AP simulator

Because of the difficulty of understanding associative algorithms, the *WebAP*, a web tool, was developed to easy the study of associative operations (Link). It provides a cycle-by-cycle simulation of an ADD associative operation, showing the states of all AP components for each step. The simulator was developed using the Angular framework. It is an open tool that serves as a support for teaching associative processing.

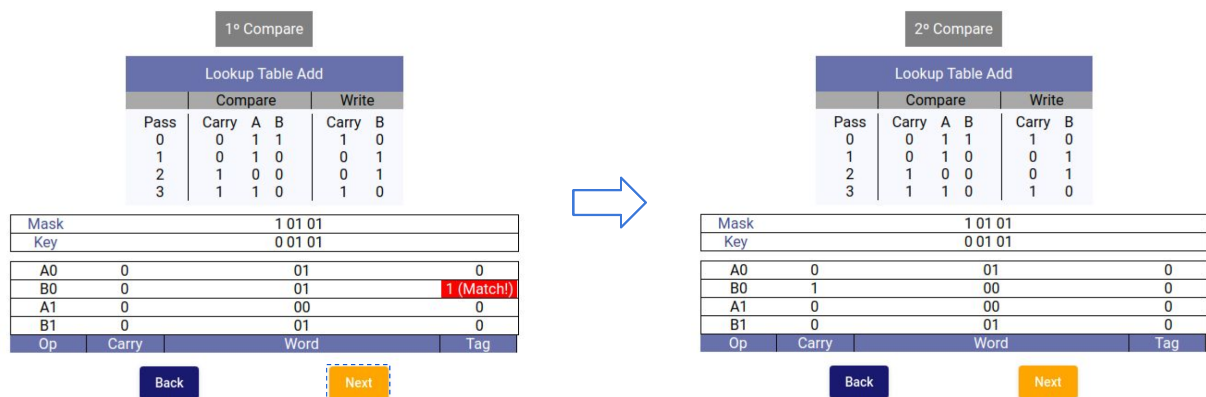

Figure 3.6: Cycle by cycle simulation on the WebAP.

To start the simulation in the *WebAP*, the user needs to set the word size of CAM, the number of values inside CAM, and, therefore, fill with values the CAM. Starting the simulation, two tables and two buttons appear, as shown in Figure 3.6. The first table represents the Lookup table and the second the data in the CAM. The buttons are used to access previous and later cycles of the simulation to analyze the algorithm step by step. The next chapter will deal with a simulator, the RV-Across, which also implements these functionalities in more operations.

Chapter 4

Related Work

Processing in Memory (PIM) is an approach that increases hardware performance by putting computations inside the memory. The PIM concept can be implemented in different ways. One of them is using a 3D memory and its reading and writing capabilities in parallel to perform logical and arithmetic operations. Another one is using the associative processing, which, through a *Content-Addressable Memory* (CAM), can compute employing comparisons and writings. Moreover, other alternatives can add features or processing units within the memory to do operations.

Research involving Processing in Memory has been increasing in the last years, mainly because of its efficiency in performing big-data applications. In the literature, our work has an intersection with PIM architectures, accelerators, programming interfaces for PIM, and simulators. They can be associated with 3D memory, Associative Processor, or other alternatives to do PIM that were developed for a general or specific purpose, evaluating latency and/or energy. In this Chapter, we start with the alternative approaches, passing through 3D memory works and finalizing with AP.

4.1 Processing in Memory

Gokhale et al. [17] in the '90s proposed a PIM model like a *Single Instruction Multiple Data* (SIMD) near memory, with an array of PIM chips, each one containing 2 kb and 64 *Arithmetic logic units* (ALUs) for computing. *Data-Intensive Architecture* (DIVA) was created by Drapper et. al. [14]. It unifies a set PIM chips as co-processors to a conventional microprocessor, using specific instruction to communicate among them, just like the Gokhale's model. DIVA has high bandwidth interconnection to be able to supply the multimedia applications demands.

Chi et. al. [8] proposed the PRIME architecture to improve the performance of NN applications utilizing ReRAM. Unlike the previous ones, PRIME has a specific purpose. The PRIME has full functions accelerators to perform search and convolution in memory, it achieves an energy relative improvement of 895x compared to other neural processing units. Shafiee et. al. [46] modified eDRAM, adding accumulation and multiplication operations within memory, to create the ISAAC accelerator. ISAAC has the power to perform matrix multiplication, accelerating the performance of NN

processing. Seshadri et. al. [45] exploited the analog operation of DRAM technology to perform bitwise operations (AND, OR, and NOT) without data movement.

The GenPIM architecture model was developed by Imani et. al. [19] to be a PIM general-purpose solution, providing memory banks capable of performing logical, searching, adding, and multiplying operations. GenPIM is composed by conventional cores linked to a main memory containing banks of PIM blocks. The authors of PIM abstracted the technique utilized. The NNPIM architecture supports the same operations as GenPIM, but explaining the mechanism of computing the operations and focusing on NN applications. The addition is inspired in Wallace-tree, the multiplication is performed through Bernstein's Algorithm and the search operation works like a CAM, using hamming distance.

PIM-Aligner is an accelerator that was made to execute DNA short read alignment. PIM-Aligner has its sense-amplifiers customized to perform XNOR, for comparison, and addition, for counting, in parallel [4]. The same group of researchers, using the same way of PIM, developed AlignS platform to execute DNA alignment in *Spin-Orbit Torque Magnetoresistive Random-Access Memory* (SOT-MRAM) [3].

4.2 3D Memories

Among the works featuring 3D memories, TOP-PIM is an architecture that is composed by 3D stacked DRAM replacing GPU and using the same interface for simulation [59].

Ahn et al. [1] proposed a programming model using custom instructions to trigger PIM in HMC (Hybrid Memory Cube). They show how to program a real application using their framework and prove low overhead generated .

The XNOR-POP architecture, created by Jiang et. al. [24], has support to PIM in Wide-IO2 DRAMs (3D memory) to process binary CNN. The Tetris accelerator follows the same line of the processing inside 3D memory, partitioning all steps of NN processing to accelerate its execution [16].

GraphP and GraphH are PIM architectures that utilize 3D memory capabilities to accelerate the processing of graph and both are HMC-based [12, 60]. Both created mechanisms in software (barriers) to facilitate the programming of search algorithms. Kim et al. proposed an algorithm called GRIM-Filter to accelerate read mapping (filtering) by reducing the number of required alignments using a 3D-stacked memory and its features [29]. Finally, Santos et al., built a framework which uses new instructions, to experiment with IoT applications using PIM in 3D memory, varying the architecture and technology of memory [44].

Several simulators have been developed for experimentation in the context of 3D memories. Leidel and Chen [31] built HMC-Sim, a cycle-accurate simulator that 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. Besides, the extension of HMC-Sim enables the users to craft Custom Memory Cube (CMC) operations and the evaluation of their efficacy through user applications. The

HMC also provides a discrete tracing allowing users to see exactly how and where memory operations progressed through the device or devices.

Jeon and Chung [23] developed CasHMC which 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. For each HMC transaction, the simulator generates a log like HMCSim.

Clapps is an HMC simulator as well, made by Oliveira et. al. [38] using SystemC. The Clapps has an interface through specific instructions for the users to perform vector PIM operations in applications. That simulator is a parallel simulator that implements all of the HMC instructions.

PIMSim and PIM-gem5 are PIM simulators with HMC that share many characteristics [44, 50]. 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.

4.3 Associative Processors

Most of the Associative Processor works have been combining NVM due to its low-overhead of area and energy consumption. Imani et al. and Yantir developed an approximate *Resistive Content Addressable Memory* (ReCAM) and evaluated [20, 51]. Imani et al. designed an accelerator model, the DigitalPIM, using associative processing to big-data applications [22].

Yantir et al. explored approximate techniques such as bit trimming and power scaling to save energy. Also, the authors explained how works an associative operation; however, they did not show how to construct the algorithms in real applications. [52, 53].

PRINS is an architecture based on ReCAM to be a programmable CAM, providing instructions and algorithms of real applications (Graphs and Machine Learning) [57]. Saikia et al. implemented a CAM with *Static Random-Access Memory* (SRAM) to accelerate the KNN algorithm and achieve 1.2 GHz of frequency spending low energy [43]. Finally, the ReSQM architecture combines arrays linked to ReCAM accelerators to do database operations [32].

4.4 Simulators

There are also works in the literature involving CAM simulators or associative processing. Khoram et. al. [28] developed an analytical model to analyze run-time, 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, the NV-Sim simulator is adapted as a tool that estimates the performance, area, and energy of CAM and other types of NVMs [33]. Yantir et. al. [51] proposed a methodology that combines approximate computing and associative processing and developed an in-house simulator for associative operations. Yavits et. al. [54, 55] also developed an associative processor in-house simulator, in which the associative processor is at the last cache level. It also estimates energy based on associative processing events.

Finally, Santos et al. [41] designed *SIM^{2}PIM*, an agnostic PIM simulator, based on the Gem-5, which implements hardware performance counters and multi-thread support for PIM devices.

4.5 Comparison

We designed an architecture model that contained a RISC-V core that can send commands to an *Associative Processor* (AP) module via custom instructions provided by RISCV instructions set. With the instructions, it is possible to load the AP with a set of data and trigger logic/arithmetic operations in parallel. This AP is linked to main memory via *Direct Memory Access* (DMA) and therefore spending fewer cycles to access the data. Thus, the way of computing in memory of our work is different from the first two groups. The custom instructions and the framework that we built are similar to works as DIVA, PRIME, TOP-PIM, and others. Our model has a general-purpose like Ambit, GenPIM, TOP-PIM, DigitalPIM, etc. Besides, we provide software support, explaining how to utilize and apply in real applications.

Some works, such as GenPIM, NNPIM, Tetris, and GraphP, create an architectural model but do not build a programming model. Others, as the architecture of Gokhale and DigitalPIM, have no assessment of energy or latency. We created a simple evaluation model for latency and energy of our architecture and with it, we evaluate a set of application kernels, even in different AP sizes.

None of the studies cited in this section used RISC-V as the base *Instruction Set Architecture* (ISA). Our work uses custom instructions, which are already geared towards acceleration, as communication support for the Associative Processor.

Most of the works targeting PIM implement are in-house simulators. This methodology might hurt the overall productivity, and it can also preclude replicability [38]. This is one of the reasons that we built RV-Across. RV-Across is an open 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. With the events counted by the simulator, the user 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 an associative algorithm, allowing operations between vectors. However, our tool offers the freedom to build and experiment with

Table 4.1: Summary of related works, classified by what was produced, the PIM approach used, the purpose and what was evaluated.

	Object				Approach			Purpose		Evaluation	
Works	**Sim**	**Sw**	**Acc**	**Arch**	**3D**	**AP**	**Alt**	**Gen**	**Spec**	**Lat**	**En**
Gokhale [17]	-	X	-	X	-	-	X	X	-	-	-
DIVA [14]	-	X	-	X	-	-	X	X	-	X	-
PRIME [8]	-	X	-	X	-	-	X	-	X	X	X
ISAAC [46]	-	-	X	-	-	-	X	-	X	X	X
Ambit [45]	-	-	X	-	-	-	X	X	-	X	X
GenPIM [19]	-	-	-	X	-	-	X	X	-	X	X
NNPIM [18]	-	-	-	X	-	-	X	-	X	X	X
PIM-Aligner [4]	-	-	X	-	-	-	X	-	X	X	X
AlignS [3]	-	-	X	-	-	-	X		X	X	X
Top-PIM [59]	-	X	-	X	X	-	-	X	-	X	X
Ahn [1]	-	X	-	X	X	-	-	X	-	X	X
XNOR-POP [24]	-	-	-	X	X	-	-	-	X	X	X
Tetris [16]	-	-	X	-	X	-	-	-	X	X	X
GraphP [60]	-	-	-	X	X	-	-	-	X	X	X
Grim-Filter [29]	-	X	-	-	X	-	-	-	X	X	-
IoT PIM [44]	-	X	-	X	X	-	-	-	X	X	-
GraphH [12]	-	-	-	X	X	-	-	-	X	X	X
HMCSim [31]	X	-	-	-	X	-	-	X	-	X	-
Clapps [38]	X	-	-		X	-	-	X	-	X	-
CasHMC [23]	X	-	-	-	X	-	-	X	-	X	-
PIM-Sim [50]	X	X	-	-	X	-	-	X	-	X	X
PIM-Gem5 [40]	X	X	-	-	X	-	-	X	-	X	X
Imani [21]	-	-	-	X	-	X	-	X	-	X	X
Yantir Approx [51]	-	-	-	X	-	X	-	X	-	X	X
PrinS [57]	-	X	-	X	-	X	-	X	-	X	X
KNN Acc [43]	-	-	X	-	-	X	-	-	X	X	X
DigitalPIM [22]	-	-	X	-	-	X	-	X	-	-	-
ReSQM [32]	-	X	-	X	-	X	-	-	X	X	X
Yantir Sim [51]	X	-	-	-	-	X	-	X	-	X	X
Yavits Sim [56]	X	-	-	-	-	X	-	X	-	X	X
NV-Sim [33]	X	-	-	-	-	X	-	X	-	X	X
Khoram Sim [28]	X	-	-	-	-	X	-	X	-	X	X
Sim^{2}PIM [41]	X	X	-	-	-	X	-	X	-	X	X
Our work	X	X	-	X		X		X	-	X	X

Sim: Simulator, **Sw**: Software interface, **Acc**: Accelerator, **Arch**: Architecture,
3D: 3D memory **AP**: Associative Processing, **Alt**: Alternatives, **Gen**: General,
Spec: Specific, **Lat**: Latency **En**:Energy.

customized operations. Our tool delivers an interface to high-level programming (via RISC-V custom instruction) and enables experiments to evaluate latency and energy. In terms of architecture, RV-Across uses a scratch-pad memory and does not consider memory coherence. 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. *SIM^{2}PIM* is a simulator similar to RV-Across that has the advantage of providing multi-thread support for the PIM device, which RV-Across does not yet provide. However, RV-Across provides in addition to agnostic support, a cycle-by-cycle simulation of an AP and a model for evaluating latency and energy that can be adapted for other architectural models. *SIM^{2}PIM* is extensible like our simulator, but by using native instructions from RISC-V, the flexibility to customize in RV-Across becomes easier. For better visualization of the related works, two tables were made. Table 4.1 refers to all works related to ours and Table 4.2 to simulators similar to RV-Across.

Our work goes beyond RV-Across. We provide case studies with associative application kernels, showing how AP works to perform them and how to implement these

Table 4.2: Selection of PIM simulators that have similarities with RV-Across.

		Structure				**Evaluation**	
Works	**Approach**	**Interface**	**Architecture**	**Algorithms**	**Extensible**	**Latency**	**Energy**
HMC-Sim [31]	3D	X	-	X	X	X	-
CasHMC [23]	3D	-	X	-	-	-	-
Clapps [38]	3D	X	-	-	X	X	-
PIM-sim [50]	3D	X	X	-	X	X	X
PIM-gem5 [40]	3D	X	X	-	X	X	X
Khoram [28]	AP	-	-	-	-	X	X
NVSIM [28]	AP	-	-	-	-	X	X
Yavits [56]	AP	-	X	-	-	X	X
Yantir [51]	AP	-	-	X	-	X	X
SIM^{2}PIM [41]	AP	X	-	-	X	X	X
Our work	***AP***	*X*	*X*	*X*	*X*	*X*	*X*

kernels (Chapter 6). Furthermore, we simplify the performance evaluation by developing experimentation models. Using these models, we simulate the kernels to highlight the potential of the AP model over CPU and multi-CPU models (Chapter 7).

Chapter 5

RV-Across: An Associative Processing Simulator

The current Chapter describes the purpose of the RV-Across simulator, its structure, the implemented architectural model, the access interface for the simulator, the operations supported, and the output generated from the simulator. Also, the latency and energy consumption models developed and the basis in the literature for the construction of the evaluation models are explained. The models served as a basis for the generation of results and conclusions of the work, which are in Chapters 7 and 8.

5.1 Overview

RV-Across is a high-level simulator for the 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 mean to calculate latency and energy. That is, the simulator is not just an addition of an extension, it is the modification of the Spike to enable the simulation of the processing in memory behavior.

Figure 5.1 shows an overview of the simulator's structure with its inputs and outputs. As input, the simulator receives the RISC-V binary that can be generated from the compilation (RISC-V tool-chain) of a C code using the library that we created to perform the operations. The output is the logs with each cycle of the CAM and the states of the Mask and Key registers in addition to showing the number of comparisons, writings, matches, and mismatches that occurred during the processing.

RV-Across is based on an associative processing architectural model to support PIM. Figure 5.2 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 Accelerator), that provides control support

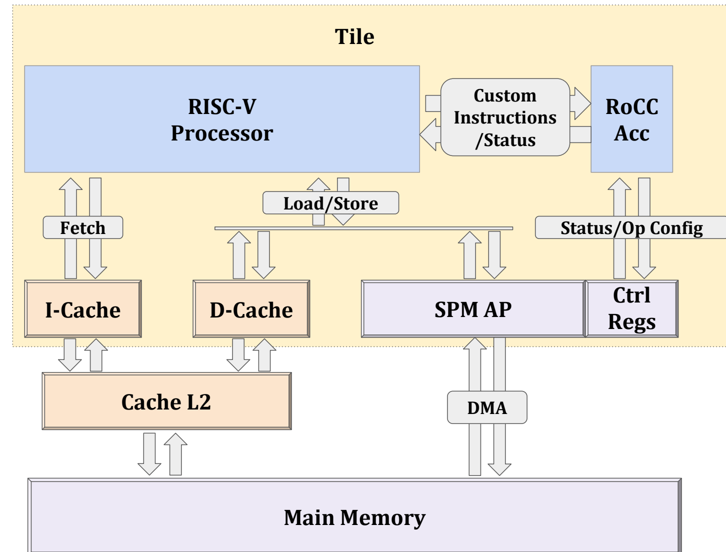

Figure 5.2: RV-Across architectural model.

for associative operations. The main core communicates with the RoCC Accelerator using custom instructions.

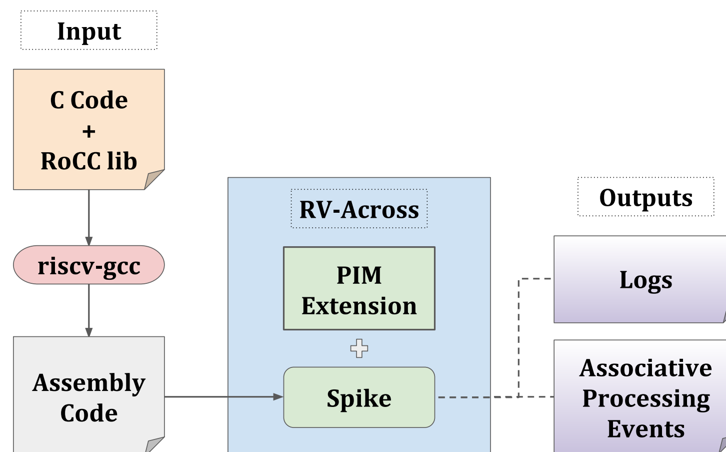

Figure 5.1: RV-Across design flow.

The RoCC Accelerator, 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 will 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 receives commands to load data from memory or trigger logical or arithmetic operations previously implemented. The complete process for associative processing is using these two steps in sequence: loading the data to be operated and finally triggering the execution at the determined addresses. The AP contains the Lookup-Table 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 *Direct Memory Access* (DMA). 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 the area can be high. For most scenarios, an AP of size 16 KB was used, because in that size there is a lower cost of area and power for the CAM. However, the execution of application kernels in different AP sizes was also evaluated to understand the impact of CAM power on the system (Chapter 7).

5.2 User interface

RISC-V RoCC instructions are used to communicate with the accelerator module attached to the RISC-V processor. Such instructions are used in the RV-Across for communication with the accelerator module that configures associative operations. Then, for the use of associative operations, it is necessary to insert the RoCC 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 instruction to use, the value for the registers, and the operation code.

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

Figure 5.3: RoCC instructions library.

The code in Figure 5.3 shows the shape of the generic macro ROCC_INSTRUCTION, which 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 operation is triggered after two steps: 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 the 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

Table 5.1: Number of comparison passes for associative operations implemented in RV-Across to a word size of n bits. The variable k means the size of the vector copied.

Associative operations	Number of passes
Unsigned multiplication	$4 \times n^2$
Addition, Subtraction	$4 \times n$
XOR	$2 \times n$
AND, OR, NOT, Shift Right and Left	n
RELU	1
SET	1
COPY	k

extension accelerator coexists 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.

RV-across provides a library implementing associative logical and arithmetic operations using the algorithm explained in Chapter 6. Table 5.1 shows the associative operations implemented in our simulator and the number of execution passes they require. 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 depends on the result of the comparisons. The sum of the number of passes and writes is the number of execution cycles needed to conclude the operation. Hence, the larger the size of the word operated, the more cycles are spent because associative processing works on comparing columns.

In addition to standard logic/arithmetic operations, SET and COPY operations were implemented in the associative processor to assist in the implementation of the algorithms. The SET operation works like the *memset*, that is, assigning a constant number to a vector. In the case of the AP, this is done in parallel due to the support offered by CAM. The same happens with the COPY operation, which is the same as the *memcopy* function, which copies multiple elements from one vector to another vector. The SET operation takes 1 cycle to configure a single write in parallel and COPY uses the number of elements to be copied. Then, with associative processing, it is possible to create new procedures that run in parallel.

5.3 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 for 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. Figure 5.4 shows the output file *ADD_1* generated from the simulation of the kernel checksum. RV-Across outputs have this format *OP_ID*, name of the operation followed by an identifier. The image shows the last cycle executed in

```
# -------------------------------------- #
    Bit: 7 | Pass: 3
    Word: 2 | Tag: 0
----------------------------------------
    Mask_A: 10000000
    Mask_B: 10000000
    Key_A: 00000000
    Key_B: 00000000
----------------------------------------
mem[0]: 11010101
mem[1]: 11100110
mem[2]: 01100101
mem[3]: 01100011
mem[4]: 01101000
mem[5]: 00000000
# -------------------------------------- #
| ----- Report ----- |
- Operations: 3
- Cycles: 40
- Comparisons: 96
- Writings: 8
-----------------------------
```

Figure 5.4: Output of RV-Across with the last cycle of the associative addition operation and the report of that operation.

the associative addition operation and a report describing the number of operations already performed, the number of cycles, the number of comparisons for each memory line, and the number of writes. Note that in order to show the column of the bit which is being analyzed, it is printed the current bit, the pass, the word operated and the tag. To facilitate debugging, the masks were separated for each vector, *Mask_A* for vector A and *Mask_B* for vector B. The key register is shown in the same way as the mask register. In the memory vector, the first half [0,1,2] indicates the elements of vector A and the other indicates vector B.

In the reports, notice the number of cycles that comprise 4 (*passes*)×8 (*word size*) = 32 cycles, as shown in table 5.1. In the results, the numbers reported by RV-Across, as shown in the figure, were used in the models that will be explained in the next section to evaluate latency and energy consumption in the execution of several application kernels in a system that uses associative processing. These results will be compared with a system that uses only the CPU to analyze the potential of the AP (Chapter 7).

5.4 Latency model

For latency evaluation, we created a formula to calculate the model latency with only the CPU and another for the model adding the AP, based on the explained architectural model of the RV-Across. The CPU latency represented by the variable L_{CPU} is the sum of the number of CPU cycles (C_{CPU}) and the cycles waiting for the data from the cache (C_{Cache}) divided by the adopted frequency (F_{CPU}). This formula is depicted in 5.1.

$$L_{CPU} = \frac{C_{CPU} + C_{Cache}}{F_{CPU}} \tag{5.1}$$

The formula for associative processing model is the sum of the latency of the auxiliary CPU to the AP and the latency of the AP. This last one is the number of the AP operation cycles added with the number of waiting cycles for DMA, divided by the AP operation frequency. Note that C_{CPU} is not $C_{CPU_{AP}}$, as one belongs to the CPU-only model and the other to the AP-connected model. The auxiliary CPU cycles correspond to the initialization cycles, the organization of the data that will be processed, and the commands sent to the AP. The latency formula for the AP is displayed in 5.2.

$$L_{AP} = \frac{C_{CPU_{AP}} + C_{Cache_{AP}}}{F_{CPU}} + \frac{C_{AP} + C_{DMA}}{F_{AP}} \tag{5.2}$$

Table 5.2 shown below describes all parameters and their respective values. The Associative Processor has the latencies determined by the operation explained in the simulator section, so it is all based on the work of Yantir [51] and implemented on RV-Across (Chapter 5). The frequency used for the AP, 1 GHz, is the same used for the CPU. It was based on the TCAM [5] we used and on other related works [56]. DMA latency varies depending on the technology used, so we based it on three articles to set the parameters described in Table 5.2 [11, 13, 15]. 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, bearing in mind that the AP will be activated when operating loads via DMA and performing operations. Both this model for latency and energy consumption can admit other models by changing only the parameters and assigning new values based on the new contexts.

Table 5.2: Latency model parameters.

Parameters	Description	Values
C_{CPU} *and* $C_{CPU_{AP}}$	Quantity of cycles performed in CPU	1 cycle/instruction
C_{AP}	Quantity of cycles performed in AP	Depends on the operation
C_{cache} *and* $C_{cache_{AP}}$	Total cycles of the CPU idle waiting for the Cache	L1 = 1 cycle L2 = 10 cycles Main memory = 100 cycles
C_{DMA}	Total cycles of the AP idle waiting for the DMA transfer	11 cycles (communication) + 1 cycle/data transferred
F_{CPU} *and* F_{AP}	Frequencies to CPU and AP	1 GHz

5.5 Energy model

We researched in the literature for power or energy parameters for RISC-V cores and found the comparison table between them in several parameters [58]. Table 5.3 summarizes the technology parameters, speed in terms of frequency, area, and power for the cores: Ariane, Rocket, Boom, and Shakti. However, none fit the same technology that we adopted for CAM [5], which is 32 nm. To solve this problem, we use a formula to normalize the power of the RISC-V Rocket core, equalizing with CAM technology.

Table 5.3: Different RISCV-cores in different parameters.

RISC-V Cores Comparisons [58]				
	Ariane	***Rocket***	***Boom***	***Shakti***
Tech (nm)	22	45	45	22
Speed (GHz)	1.7	1.6	1.5	800 MHz
Area (mm²)	0.3	0.5	1.7	0.29
Power (mW)	52	125	300	90

So, the new power given by the variable P_{scaled} is equal to the multiplication of: the operating frequency (F), in this case 1.6 Ghz; the division between the new (T_{new}) and the old technology (T_{old}), 32 nm and 45 nm; and the voltage supply that we consider to continue with 1 (VDD_{old} and VDD_{new}) [5]. The new power calculated via equation 5.3, matched to 32nm technology for the Rocket core, is 55.56 mW.

$$P_{scaled} = F \times \left(\frac{T_{new}}{T_{old}}\right) \times \left(\frac{VDD_{old}}{VDD_{new}}\right)^2 \tag{5.3}$$

For the Associative Processor, we searched for data on the CAM and found an article describing a TCAM, 32nm with 1GHz, implemented. This article provided a bit rate for both area, 0.84 Megabit/mm^2, and power, 0.58 W/Megabit. From these parameters, we derive power and area for different types of TCAM. We adopted these parameters as a reference for the Associative Processor considering that the overhead for implementing the algorithms and lookup tables is small.

Table 5.4: Power and area parameters derived from TCAM 32nm

TCAM - 32 nm [5]				
Size (Kb)	***16***	***32***	***64***	***128***
Power (mW)	74.24	148.48	296.96	593.92
Area (mm²)	0.152	0.304	0.609	1.219

As we did for latency, we built two equations to calculate the energy consumption of the model using CPU-only and the model using AP. Equation 5.4 represents the energy spent in the execution of applications on the CPU (E_{CPU}), which is the multiplication of the CPU power (P_{CPU}) and the execution time of the application. This last one is

Table 5.5: Power parameters derived from the literature

Parameters	Description	Values
P_{CPU}	CPU power based on Rocket core	55.56 mW
P_{CAM}	CAM power	74.24 mW for 16 kb
$P_{CPU_{LP}}$	Power of low power mode	15.4 mW

represented by the sum of the CPU cycles (C_{CPU}) and the cycles waiting for the cache, divided by the frequency (F_{CPU}).

$$E_{CPU} = P_{CPU} \times \left(\frac{C_{CPU} + C_{Cache}}{F_{CPU}} \right) \tag{5.4}$$

The equation for the AP requires complexity because when the auxiliary CPU is running, the AP is turned off, and when the AP executes an instruction, the CPU remains in low power mode. So, for the equation 5.5 we need to calculate the energy of the auxiliary CPU separately and consider the low power mode in the energy calculation for the Associative Processor. Therefore, the AP total energy (E_{AP}) is the power of the auxiliary CPU (P_{CPU}) times its latency, comprised by the sum of the execution cycles ($C_{CPU_{AP}}$) and idle ($C_{Cache_{AP}}$) over the frequency (F_{CPU}), added to the addition of the power of the CAM (P_{CAM}) plus the power of the low power mode ($P_{CPU_{LP}}$) times the latency of the CAM, represented by the sum of the AP (C_{AP}) and DMA (C_{DMA}) processing cycles divided by the frequency (F_{AP}).

$$E_{AP} = P_{CPU} \times \left(\frac{C_{CPU_{AP}} + C_{Cache_{AP}}}{F_{CPU}} \right) + (P_{CAM} + P_{CPU_{LP}}) \times \left(\frac{C_{AP} + C_{DMA}}{F_{AP}} \right) \tag{5.5}$$

To extract the power parameter for low power mode, we calculated a relative proportion between the active power and the low power found in Kumar et al. [47]. Their work implements clock gating for a Rocket core and the technique causes the core to consume 28% of the normal active power. Using the proportion with the equalized power previously calculated (55.56 mW), the low power mode is 15.4 mW. Table 5.5 summarizes the derived power values that were used for experimentation described in subsection 7.1.

5.6 Customizing associative operations

RV-Across comes with an extensible structure that allows the user to modify the existing operations or implement their own within its core. Figure 5.4 shows a simplified *Unified Modeling Language* (UML) showing the *APrv* class used in the simulations, inheriting the attributes and methods of the model class *APTemplate*. Note that the *APTemplate* class already provides everything needed for associative processing. This class 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.

The user has to create a new class like *APrv*, inheriting the attributes and methods of the *APTemplate* class. Then, to create new operations, the user just needs to describe

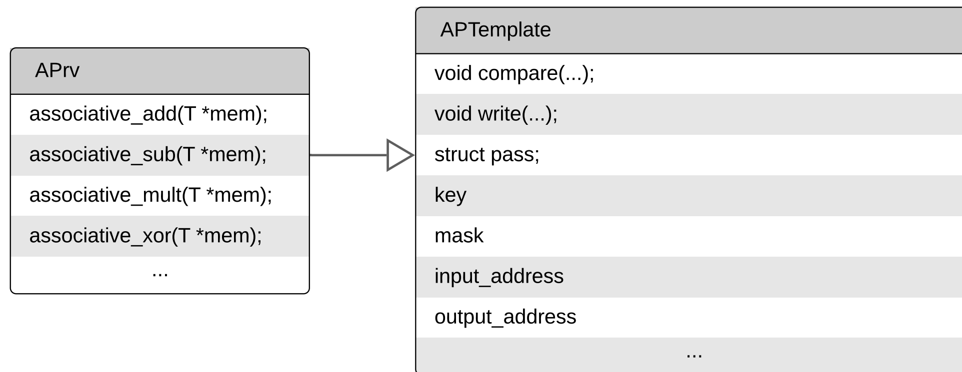

Figure 5.5: UML showing the APrv inheriting attributes and methods from APTemplate for the RV-Across simulation.

the associative algorithm in terms of these base structures. To create an associative operation, the user must follow the steps explained in the previous Chapter. After creating the operation, the developer must assign it to a specific RoCC instruction. Then, as shown in subsection 5.2, it must write a macro referencing its operation with two instructions: to load the elements of the operation vectors and to trigger the operation. Following these steps when writing the macro in C code, generating the RISC-V binary, and executing in the RV-Across, the custom operation will be performed correctly.

Besides the customized operations, RV-Across supports a trace function to analyze pass by pass, producing cycle-accurate information about the execution and aiding in the development and debugging of new associative algorithms.

Chapter 6

Case studies

After explaining what is RV-Across and how it works, this Chapter shows in practice how to implement specific kernels and their behaviors into the AP. We selected Matrix multiplication, 2D convolution and ReLu activation function, because they needed a strategy to extract parallelism from the AP. For each application, the use of the RV-Across operations and how the data is processed within the CAM is explained.

6.1 Matrix Multiplication

Matrix multiplication is essential for various machine learning and image processing applications (within computer science) [48]. It is a binary operation, in other words, two matrices becoming one. In a multiplication between a matrix A and matrix B resulting in C ($C = A \times B$), C is the accumulation of the multiplication of the rows of matrix A and the columns of matrix B. There are many ways to parallelize this operation both in the context of many processing cores and in a SIMD. Our goal is to try to explain in a simple way how to parallelize the matrix multiplication using our simulator and its software interface. We also aim to show the working inside the AP.

```
1 for(i = 0; i < DIMENSION; i++) {
2     row = i * DIMENSION;
3         for(j = 0; j < DIMENSION; j++) {
4             row_B =  j * DIMENSION;
5             SET_RVA(buff, DIMENSION, *(A + row + j), 1);
6             MULT_RVA(buff, B + row_B, buff, DIMENSION, 1);
7             ADD_RVA(C + row, buff, C + row, DIMENSION, 1);
8         }
9 }
```

Figure 6.1: Associative algorithm for matrix multiply using custom instructions for implementing row parallelism.

Giving an overview, we use row parallelism, in which we select an element of A, assuming $C = A \times B$, and in parallel, we multiply with the entire row of B and in parallel, we accumulate the entire result in C. We do this for all elements of A and lines of B

following the standard algorithm. To implement this algorithm using RV-Across and without ever losing sight of associative processing, we need to use three instructions, SET_RVA, MULT_RVA, and ADD_RVA, within two loops ranging from 0 to the size of the matrix dimension, as shown in Figure 6.1. Through SET_RVA we fill a buffer, having the size of the matrix dimension, with each element of the matrix A. Then we perform a multiplication, by MULT_RVA, in parallel between the buffer and a row of B and we save the result in the buffer. Finally, we accumulate the buffer values on the C row using the associative addition in parallel via ADD_RVA. In the traditional algorithm, without parallelism, three loops are used to iterate between the elements and computation. Thus, we readily noticed the removal of a loop in the above algorithm and consequently a loss of complexity. A simple calculation of the AP latency cost of this algorithm can be done based on the dimensions of the matrix, treating them as squares. The latency would be the square of DIMENSION times the summation of AP operations latencies.

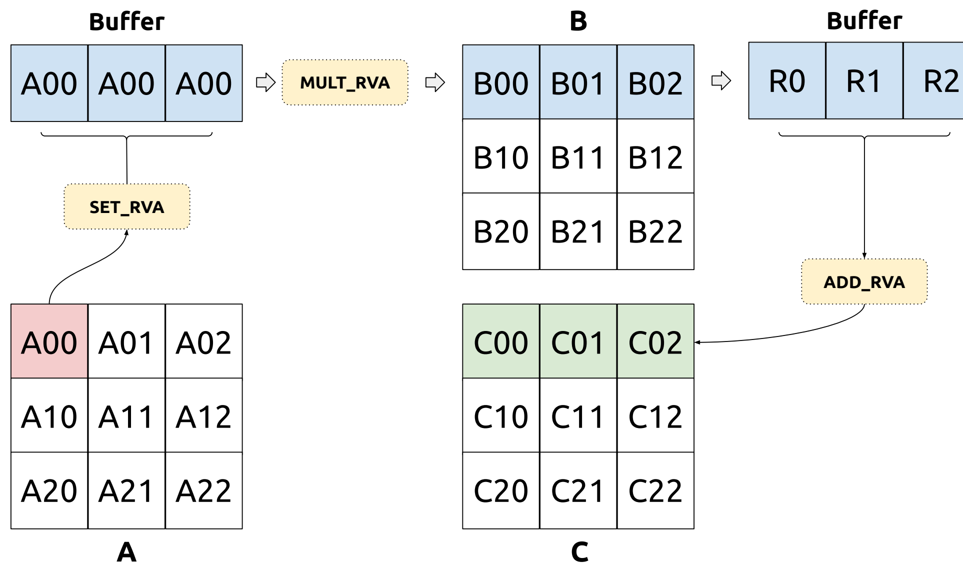

Figure 6.2: Overview of the behavior of matrix multiply in associative processing.

Looking inside the AP, now let's analyze how it works, looking at Figure 6.2. The figure shows the three matrices A, B, C, and the buffer in the process of the first iteration of the algorithm, all happening inside the AP. First, matrices A and B must be loaded assuming that the AP space has enough capacity to contain the three matrices and the buffer. After that, the AP with the base addresses of matrix A and the buffer in its special registers, performs the copy of the elements with SET_RVA, makes the multiplication with the first row of B saving in the buffer (a destructive operation), and accumulates in the reserved space inside the CAM for matrix C. Finally, all results are written to main memory via DMA.

6.2 2D Convolution

The 2D convolution operation is widely applied, as well as the multiplication of matrices, in several areas of knowledge (image processing, digital data processing, machine learning, etc.) [7]. Matrix convolution is an operation between a kernel and a data matrix (eg. image). The kernel acts as a filter applied to the image. Each element of the 2D convolution output matrix is the result of the sum of the multiplication between the kernel and the matrix, which is equal in size to the kernel, extracted from each element of the data matrix. The matrix extracted from each element is composed of all the elements around it, each element being a central reference for matrix, as illustrated in Figure 6.3. The figure highlights in green the parts of the kernel and the extracted matrix that will be multiplied and accumulated for the first index of the output matrix. Note that other elements of the kernel have been rejected due to the position of the image element. It is essential to understand how this extraction is done because it is important for understanding how we implement it in the AP.

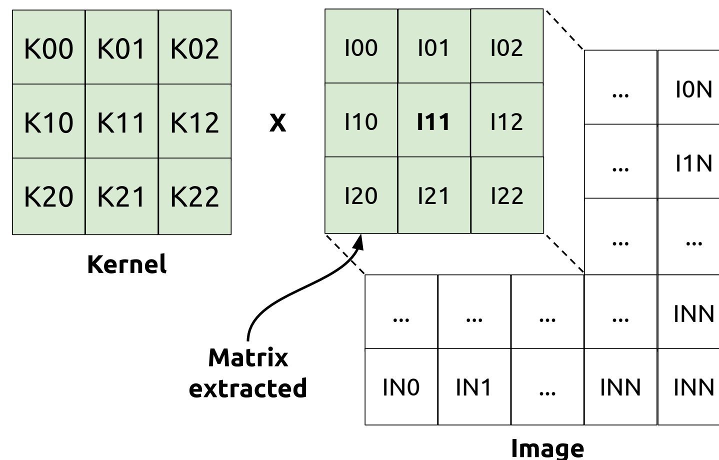

Figure 6.3: Matrix extracted from the first element of the image operating with part of the kernel.

To obtain a better performance of the AP, it is necessary to perform an associative operation on as much data as possible, and this became a problem at the beginning of the associative convolution implementations. Implementations of multiplication parallelism and accumulation of results per kernel did not perform well. For this reason, it was necessary to organize the image data to be able to increase the parallelism. The main idea of adapting to the associative algorithm was to transform the image into a new matrix to use the same flow of associative operations of the matrix multiplication. The new matrix, which in Figure 6.4 is called "Image data organized" (image-to-column), is composed of all extracted matrices explained before. All extractions, in the new matrix, are columns, thus having a dimension of the number of kernel elements as height and number of image elements as width. Figure 6.4 shows an example of what this new matrix would look like if the kernel is 3x3 in size. In this case, it would be LxM with L being the number of elements in the kernel and M being the number of elements in the image.

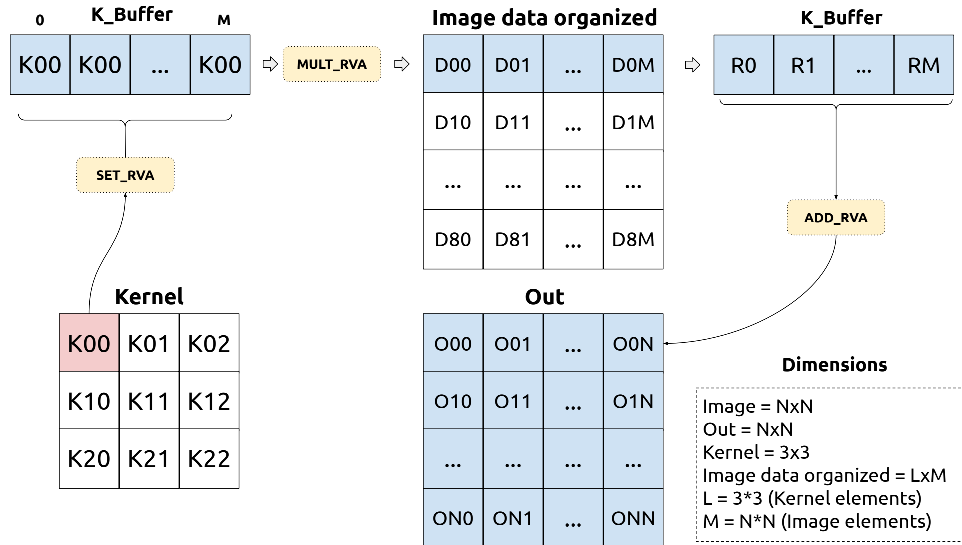

Figure 6.4: Overview of the behavior of 2D convolution in associative processing.

With the matrix organized with elements from different matrices extracted on the same line, it is now possible to parallelize multiplication and accumulation not only by extracted matrix, but among all of them. Note that in this way, convolution has the same shape of associative matrix multiplication. And for the implementation of the convolution, we follow the same steps in a similar algorithm shown in Figure 6.5. First, the buffer (*k_buffer*) is created, with the same size of the image, to receive a copy of each kernel element. This procedure is done via SET_RVA. Then, the *k_buffer* containing the first element of the kernel is multiplied in parallel, by MULT_RVA, with all the first elements of all extracted matrices, and saved in *k_buffer*. Lastly, the *k_buffer* is accumulated in the output image. This process is done with all the elements of the kernel towards all the lines of the organized matrix, reaching high parallelism guided by the number of elements of the kernel.

```
1 uint8_t k_buff[len];
2 SET_RVA(out, IMG_SIZE, 0, 1);
3 for(i = 0; i < KERNEL_SIZE; i++) {
4     SET_RVA(k_buff, len, kernel[i], 1);
5     MULT_RVA(k_buff, &image[j], k_buff, IMG_SIZE, 1);
6     ADD_RVA(out, k_buff, out, IMG_SIZE, 1);
7     j += IMG_SIZE;
8 }
```

Figure 6.5: Associative algorithm for 2D convolution using custom instructions for implementing row parallelism.

However, this algorithm becomes expensive because the new matrix introduces a high additional overhead. Besides, the cost to organize is much higher than computing in the AP. We did experiments, both organizing the data and having it already processed. With the data already processed it is possible to obtain a performance twice as good as a CPU and save twice as much energy.

6.3 ReLU

The artificial neural networks uses the activation function to characterizes the node outputs, similar to a button, associated to a circuit, which the mode "on" or "off" is chosen by the function [9, 49]. The *Rectified Linear Units* (ReLU) activation function, in the binary step, selects the output values with the simple criteria. As displayed in the Equation 6.1, if the values of the equation are greater than or equal to zero, then they will be 1, otherwise they will be 0. This very important activation procedure can be optimized by associative processing by performing a simple step.

$$\begin{cases} 1 & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases} \tag{6.1}$$

The associative implementation for ReLU takes advantage of the binary representation of the number and reduces the problem to a search for positive numbers. In integers, the signal is represented by the most significant bit, 0 for positive and 1 for negative. For the initial setup of the algorithm, the vector containing the values to be analyzed is loaded into the CAM. Then a buffer with the size of the vector is instantiated and filled with "ones" utilizing SET_RVA, as illustrated in the Figure 6.6.

```
1 uint32_t B[len];
2 SET_RVA(B, len, 1, 4);
3 RELU_RVA(A, B, len, 4);
```

Figure 6.6: Associative algorithm for ReLU using special instruction.

The buffer, represented by "B", containing "ones" serves as pre-written ReLU results, in other words, assuming that all numbers are already positive. After that, the main step of the algorithm is done by the RELU_RVA instruction. This instruction triggers an operation in the associative processor that sets its mask and key to check the most significant bit of the vector of values "A" and record of negative numbers in vector "B", writing 0 in the respective positions.

See the example represented in the Figure 6.7. Note that vector B contains pre-written ones and vector A contains $[-x, y, -1, z, -k]$. Remembering that the CAM is able to analyze all the bits of a column in a cycle guided by the special registers, mask, and key. The mask selects the most significant column for A and the least significant column for B, the key representing the comparison bits of the lookup table. The lookup table for this operation has as comparison bit 1 for both A and B and write 0 in B if there is a match, that is, if a negative number is found in A. Then, the CAM does the comparison in parallel and if it finds a negative number, as in the case of [-x, -1, -k] in A, the value 0 is written in B, as represented in the lookup table, and therefore selecting as ReLU.

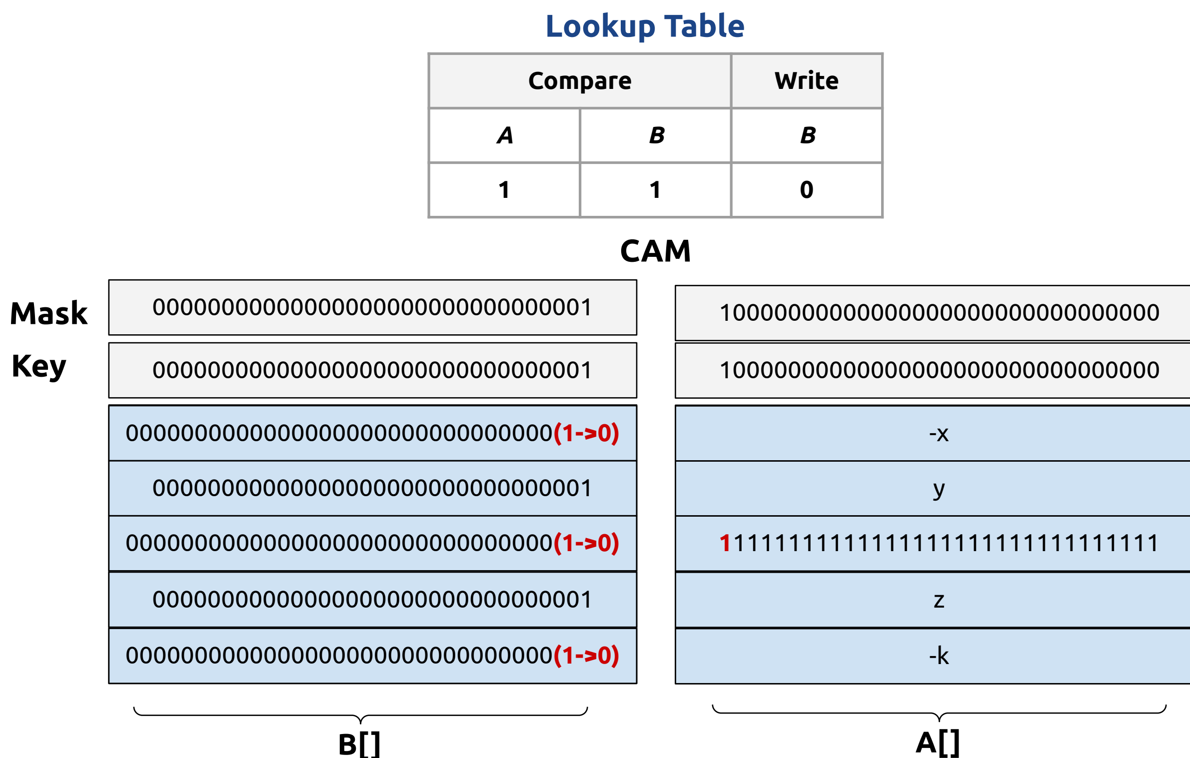

Figure 6.7: Simulation of the ReLU operation within the CAM.

The associative algorithm for ReLU so far has two limitations. The first is that this RELU_RVA operation only supports integer numbers, but because of the CAM logic, the comparison can be adapted to floating-point values, being only necessary to adapt the position of the mask and the key. As future work we intend to make the algorithm adaptable, encompassing all types. The second limitation is that a buffer of the same size as the value vector is needed to operate, which can be costly in terms of space for the CAM. On the other hand, the algorithm spends a cycle for comparison and a cycle for writing, assuming that the vectors are already organized within the AP and therefore speeding up the filtering of the values. It is worth using it both to improve performance and to save energy. The next Chapter shows all the results for matrix multiplication, 2D convolution and ReLU.

Chapter 7

Demonstration and Results

Demonstrated and exemplified all the theory on which we are based, this Chapter deals with the evaluation of kernels of applications. We evaluate our proposal in two scenarios. In the first scenario, we compare our AP approach with a conventional single-core CPU. 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 this scenario, we evaluate the execution latency, energy consumption and amount of latency, according to the models that were explained in the subsections 5.4 and 5.5. Then, in the second scenario, we show a performance comparison between the AP and a multi-core 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. The evaluation was made with data reported from the RV-Across and applied to the models. Finally, we evaluate the simulation performance of the RV-Across.

7.1 AP vs CPU

A total of eight application kernels were implemented for experimentation: Bitcount, checksum, 1D convolution, hamming distance, manhattan distance, ReLU, matrix multiply, and 2D convolution. The applications were evaluated in the context of a model using only the CPU versus a model with an Associative Processor, applying the latency and power models explained in Chapter 5. A graph was generated for each application with the comparison of latency, energy consumption, and quantity of load/stores executed in order. These comparisons were made by varying the size of the input of each algorithm respecting the 16 KB size of the CAM adopted for this evaluation. Each entry depends on the algorithm, for example, in the case of the matrix multiplication algorithm, the entry is the dimension, in the case of the checksum algorithm, the entry is the quantity and bytes of the array. For some algorithms, it is possible to fill the entire CAM, for others it is not possible, due to the overhead of space used. The algorithms were implemented and evaluated for execution with data in bytes. Matrix multiplication and convolution were also deeply evaluated, varying the size of the

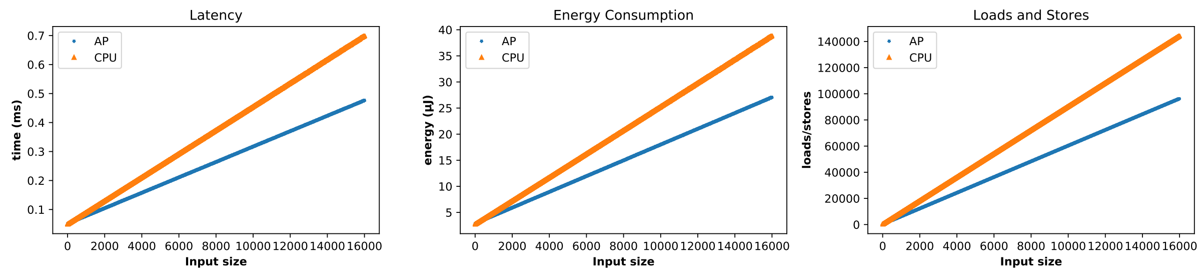

Figure 7.1: Latency, energy consumption and memory accesses for AP and CPU, both performing bitcount.

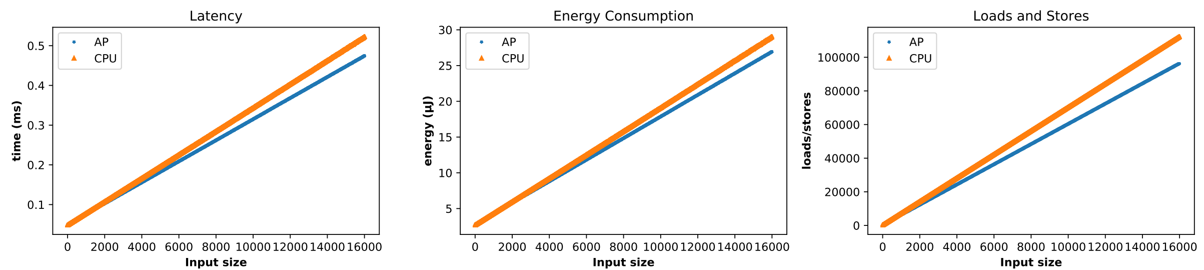

Figure 7.2: Latency, energy consumption and memory accesses for AP and CPU, both performing checksum.

AP and energetically evaluating the impact of a high power context. Also, a graph was generated to explain the organization overhead used for 2D convolution and the result of an optimized alternative. Finally, we plotted a graph for an overview of the relative improvement and its geometric mean. Note that for almost all cases, the gain is proportional to the size due to the relationship between the number of elements to be parallelized and the execution overhead of the AP.

Figures 7.1 and 7.2 show the execution latency, energy consumption and the amount of load and stores for CPU and AP executing bitcount and checksum respectively. The checksum algorithm generates verification data through the successive sum of several data. The bitcount returns the number of bits (ones) in a list of bytes. So, both algorithms need to accumulate data and counts. For this, a similar mechanism for summing the AP was used for both. 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 is computed in parallel, and then the result is accumulated using the checksum algorithm. Both experiments filled the CAM, 16 KB. Among them, the bitcount obtained a greater relative gain over the CPU. The checksum is an application already executed quickly by the CPU, for the simplicity of being an accumulation. Besides, with the AP it is not possible to make a sum between all the data array at the same time. The maximum that the CAM can achieve is to perform a sum between halves, treating the halves as vectors, as explained in Chapter 3. In both applications, the tie point with the CPU was around the size of 200 bytes of input. The bitcount execution in the AP was 1.4x better in all aspects shown in the graphics and the checksum was 1.1x better. The relationships were made for the maximum input sizes for each algorithm.

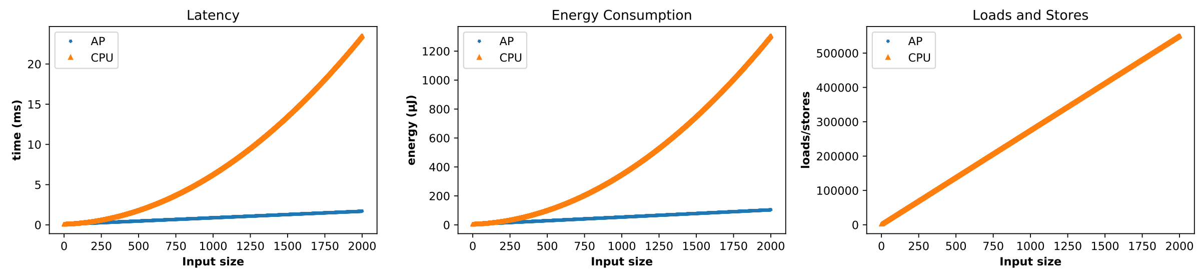

Figure 7.3: Latency, energy consumption and memory accesses for AP and CPU, both performing 1D convolution.

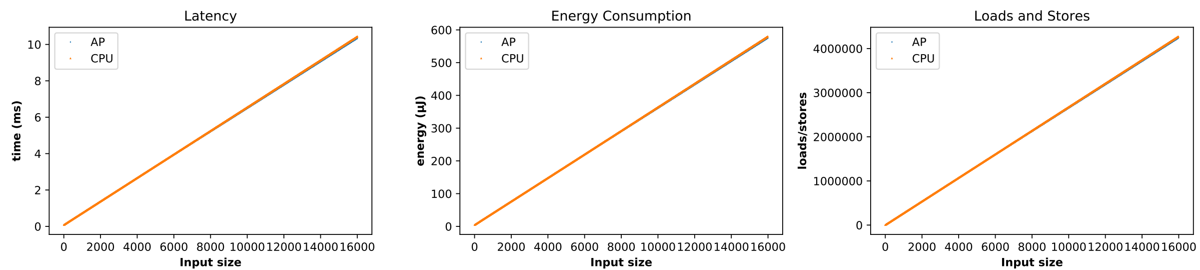

Figure 7.4: Latency, energy consumption and memory accesses for AP and CPU, both performing hamming distance calculation.

In the results for 1D convolution, shown in the Figure 7.3, the behavior of the load/store metric differed from the previous applications described. 1D convolution is performed between vectors, with the same principle that a kernel vector operates on all elements of a vector data set, with each element of the data vector as a reference. The algorithm for using vectors facilitates adaptation to the associative algorithm. However, the operations between the kernel and each element of the vector, in this case, require the CPU to assist the AP with many accesses to memory. Thus, creating a scenario of great processing potential for the AP model, at the same time that the auxiliary CPU is greatly required. The results for latency and energy show the gain related to size, as in other cases. For latency, a gain of 13x and, for energy, of 12x. However, the algorithm spends a lot on the use of loads and stores tying with the CPU model.

Unlike the 1D convolution, the results of the evaluation for hamming distance, displayed in the Figure 7.4, were tied for the three metrics. The hamming distance is a byte comparison algorithm, generally used for strings, that is a vector of bytes. The main operation is XOR, the standard logical operation for differentiating between bit-level values. In short, in the AP it was necessary to make only one XOR among all the elements, while in the CPU an XOR was executed for each byte. However, the execution speed on the model with the CPU is so high that it is not worth running on the AP, as the improvement is low. Even filling an entire CAM and running the XOR in parallel, the relative gain appears in the third decimal place.

When experimenting with the Manhattan distance kernel in the two models, as shown in figure 7.5, it is noticeable that the application performed well in the AP model. The distance from Manhattan is the value of calculating the sum of the differences between coordinates. The application gives the data corresponding to the coordinates

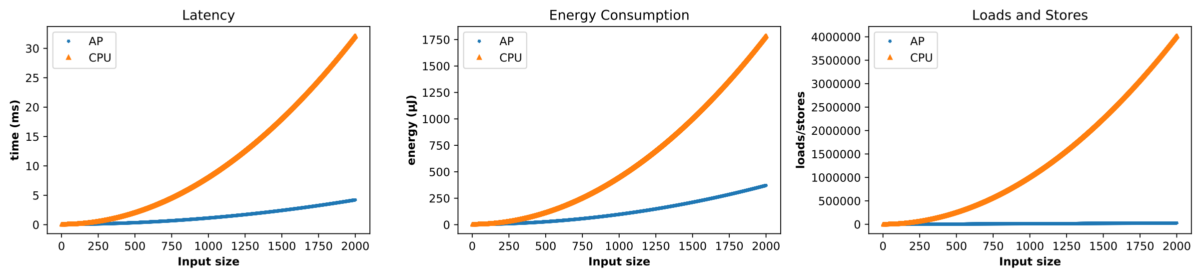

Figure 7.5: Latency, energy consumption and memory accesses for AP and CPU, both executing Manhattan distance.

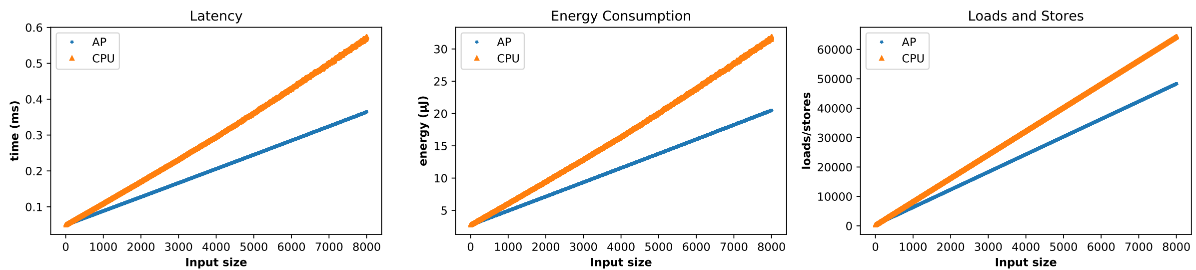

Figure 7.6: Latency, energy consumption and memory accesses for AP and CPU, both executing ReLU.

well-organized in lists. It favors the associative operation that has the power to perform additions and subtractions in parallel, while the CPU model needs to perform two loops to iterate over the coordinates and perform the calculation. The distance from manhattan running on the AP model achieved a relative gain of 7x for latency, 4x for energy, and 138x in the quantity of load and stores (the best among applications).

Figure 7.6 shows a comparison between the CPU and AP models when performing the ReLU activation function. ReLU is an activation function that defines the outputs of the nodes of a neural network. It assigns 1 for numbers that are greater than or equal to zero and otherwise it assigns 0. Special instruction for the selection of elements within the AP was implemented in the AP as a ReLU selection. The instruction searches in the column of data, whose values are negative through the most significant bit, and writes in parallel to the result vector. The complete explanation can be found in subsection 6.3. The relative improvement of ReLU running on the AP was 1.5x for latency and energy and 1.3x for load/store quantity.

The matrix multiplication kernel was also evaluated in terms of latency, energy consumption, and quantity of load/stores in the CPU models and with the AP. The result is illustrated in 7.7. In adapting to extract parallelism from the context of associative processing, we use row parallelism, where we copy each element of a vector and replicate it in a buffer. Then, in parallel, we perform both multiplication and accumulation in the output vector. So, we removed a loop from the standard algorithm for performance. In subsection 6.1, both the algorithm and the behavior within the CAM are explained in detail. The superiority in performance is clear in the numbers of relative improvement: 6x for latency, 4x for energy consumption, and 28x for load/stores.

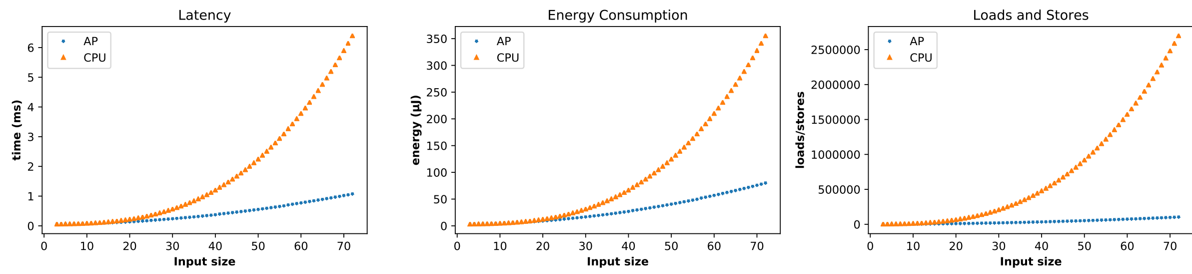

Figure 7.7: Latency, energy consumption and memory accesses for AP and CPU, both performing matrix multiply.

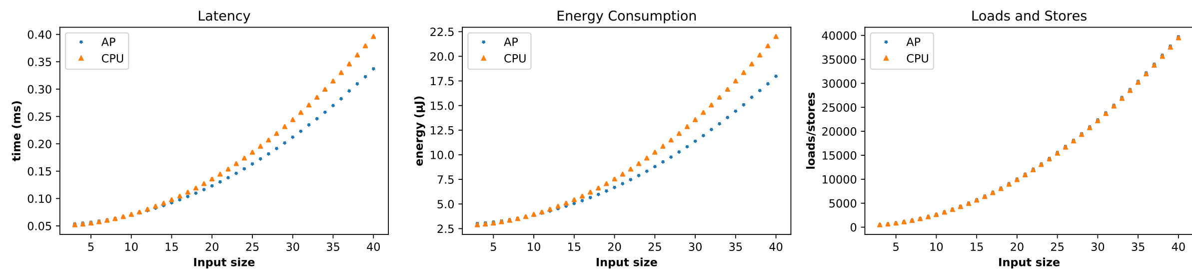

Figure 7.8: Latency, energy consumption and memory accesses for AP and CPU. The CPU performing naive 2D convolution algorithm.

The results for 2D convolution with the same matrix multiplication metrics are shown in Figure 7.8. This figure shows the comparison between a naive version of the 2D convolution implementation versus the AP implementation. For the 2D convolution kernel, it was necessary to organize data overhead so that they would have an exact shape to execute the same parallelism mechanism used in matrix multiplication. The form of organization is best explained in the subsection 6.2. So, against the naive version running on the CPU, note that the AP model gains in latency and energy but ties in load/store because this organization made to extract parallelism from the AP.

However, the comparison of the version of the 2D convolution in the AP versus the software optimized version performed in the CPU model ties for latency and energy consumption, as notable in Figure 7.11. Interestingly, in this software-optimized version, memoization is used to speed up computation and this can be seen in the graph as the CPU spent a lot of load/store operations. Thus, the AP achieved a relative improvement of 1.7x.

The alternative found to improve the algorithm was to reimplement the algorithm assuming that the matrix data is already organized and ready to be operated on the AP. We did this by keeping the entire array organized in binary files. In this new version of the associative algorithm, this time optimized, the program loads the data from the file and already executes it in parallel in the AP. Figure 7.10 depicts the comparison between the normal and the optimized version, where the graph above represents the non-optimized algorithm and the one below the optimized. Notice the high expense required to organize the data. Although the initialization latency is high, it appears small in the presence of organization cost. Note also the constant computation latency in the AP that causes a positive impact on performance with increasing data size.

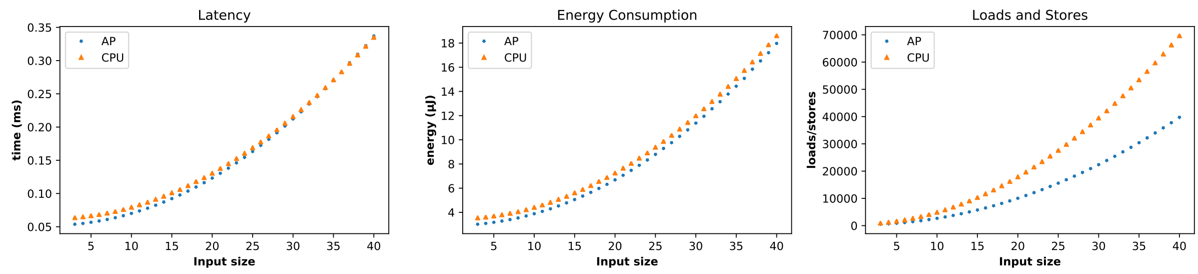

Figure 7.9: Latency, energy consumption and memory accesses for AP and CPU. The CPU performing optimized 2D convolution algorithm.

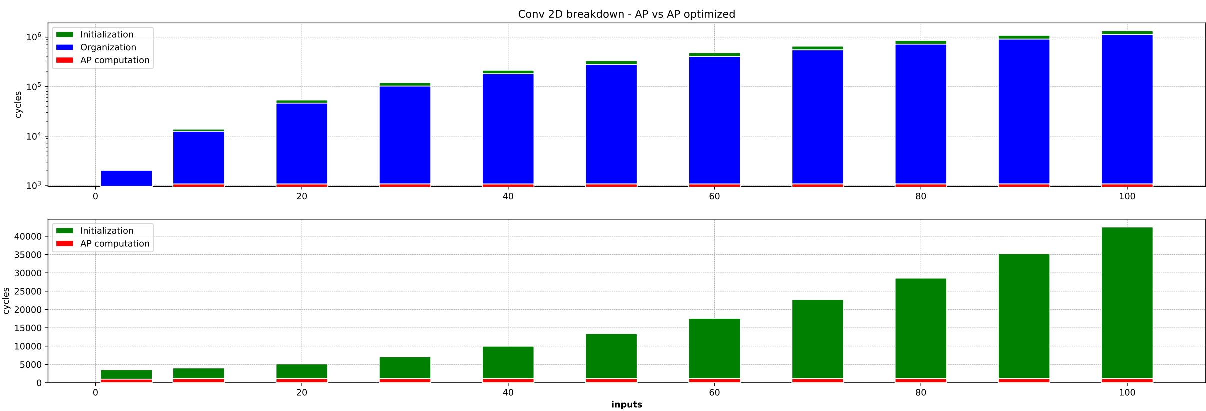

Figure 7.10: Comparison in number of cycles between the normal and optimized version of the 2D convolution in the AP, separating the executions in phases for initialization, data organization and computation in the AP.

Finally, in the figure 7.11, we compare the optimized versions for the model in the AP and the CPU. The execution of the 2D convolution optimized for the AP model obtained a relative improvement of 2x for latency, 2x for energy consumption, and 13x in load/store operations.

Figures 7.12 and 7.13 represent an analysis of the energy impact between CPU and AP models in different AP sizes, from 16 KB to 32 KB, running 2D convolution and matrix multiplication respectively. Each range of the graph corresponds to a different size of AP, being guided by the supported input. In other words, for matrix multiplication, the inputs from dimensions 3 to 72 fit into a 16 KB AP, but from 73 to 120, it is necessary a 32 KB size AP, and so on. See that as the AP grows in size, the energy consumption increases, which is due to the increase in power. In Figure 7.12, we compare the two versions of implementations of 2D convolution in the AP with the CPU model. Note that there is a relative gain of the AP in both versions for the 16 KB and 32 KB APs. But from 64 KB, the version of the AP without the optimization starts to lose and the optimized version keeps saving energy up to the maximum size of 128KB. This is because the normal version has a high overhead for organizing the data. The scenario, shown in the figure, for matrix multiplication is better concerning 2D convolution. For all AP sizes, multiplication continued to save energy. That is, regardless of the size of the AP, matrix multiplication in the associative processor is a better option for energy saving compared to a CPU only model.

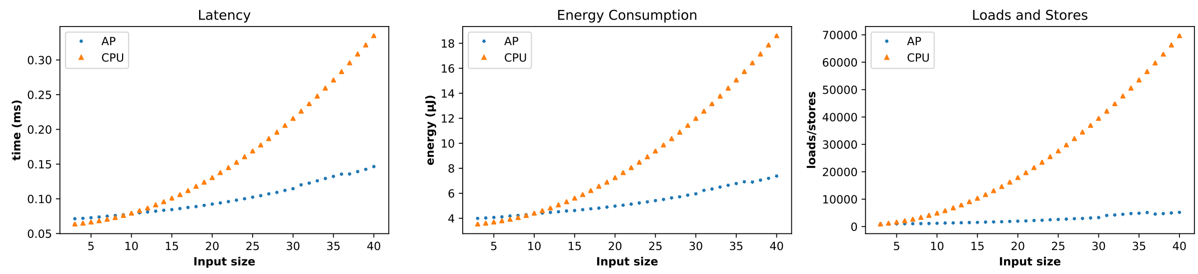

Figure 7.11: Latency, energy consumption and memory accesses for AP and CPU. The AP and CPU, both performing optimized 2D convolution algorithm.

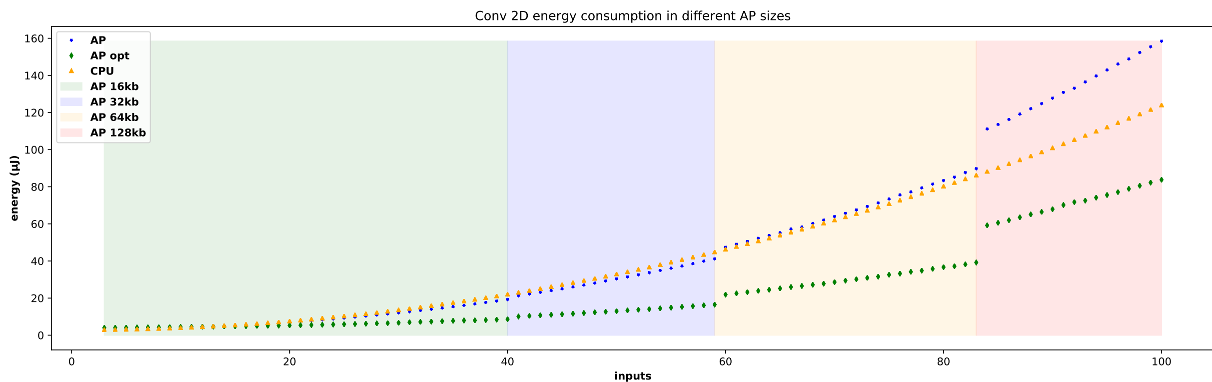

Figure 7.12: Energy consumption for different AP sizes, comparing the normal and optimized for AP and optimized version for CPU.

In chapter 3 is shown that the more the word size increases in the execution of an associative algorithm, the more it is costly to execute. This is because the associative algorithm will need to analyze more bits and write in more columns. Figure 7.14 shows the number of cycles, spent only inside AP, performed to multiply matrices in different word sizes. Note that, as expected, the larger the word size, the more cycles are required for execution. For a notion of proportion, in the case of the execution of the 100x100 matrix, running with 8 bytes of word, it is spent 36x more than running with 1 byte.

Table 7.1 shows all the relative gains and input ranges for all applications used in the experiments. In general, the model with the AP proved to be efficient in performance and energy. However, applications that are already very fast on the CPU or that require a larger organization overhead did not fit well with the model. Deriving from the results, for the model with the AP to obtain a good performance, the application must already provide a data organization or this organization must have a small overhead. The AP is more energy efficient in smaller sizes like 16KB or 32KB. Besides, a 16KB CAM has approximately half of an Ariane core area, as shown in the table 5.3, and a 32KB CAM is similar in size to an Ariane core. So it is worth having an AP that spends the same area as a core and has a performance of up to 6x of a CPU when a matrix multiplication is performed. Therefore, the AP becomes a viable alternative for embedded devices that require energy saving and at the same time processing speed.

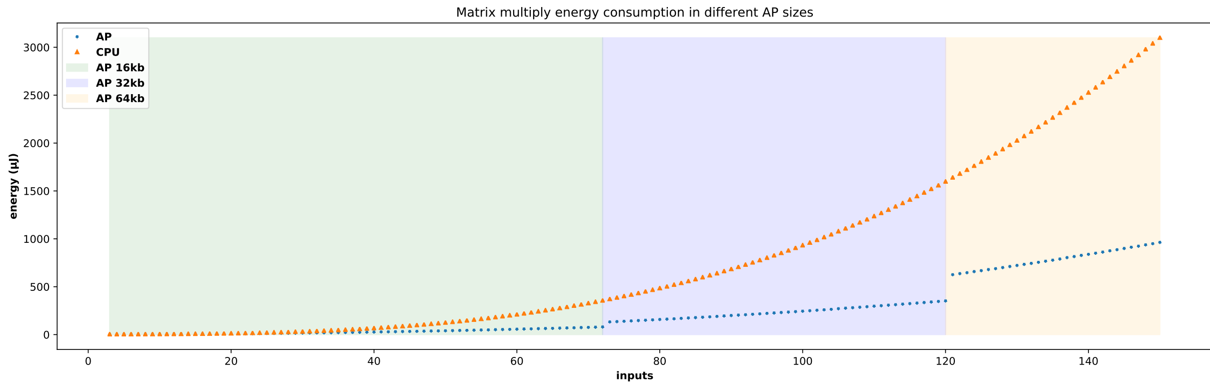

Figure 7.13: Energy consumption for different AP sizes, comparing AP and CPU, performing matrix multiply.

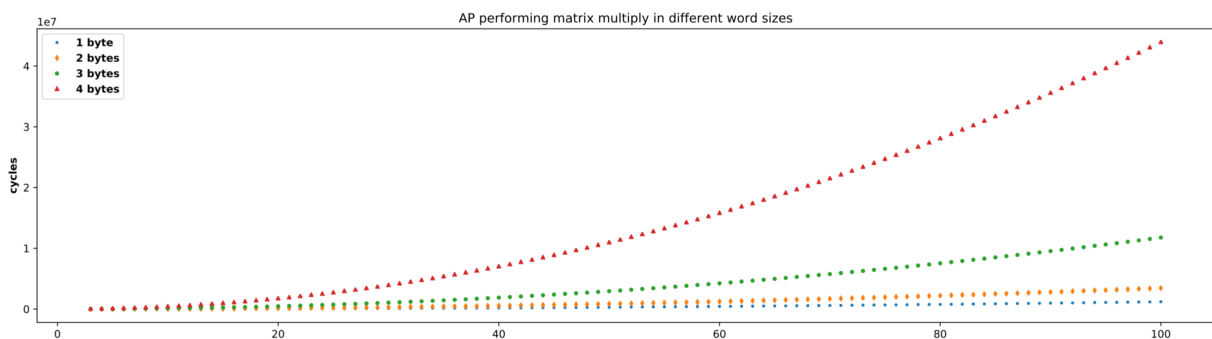

Figure 7.14: AP performing matrix multiply with different word sizes.

7.2 AP vs Multicore

In the multi-core 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. 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 7.15 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 multi-core 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

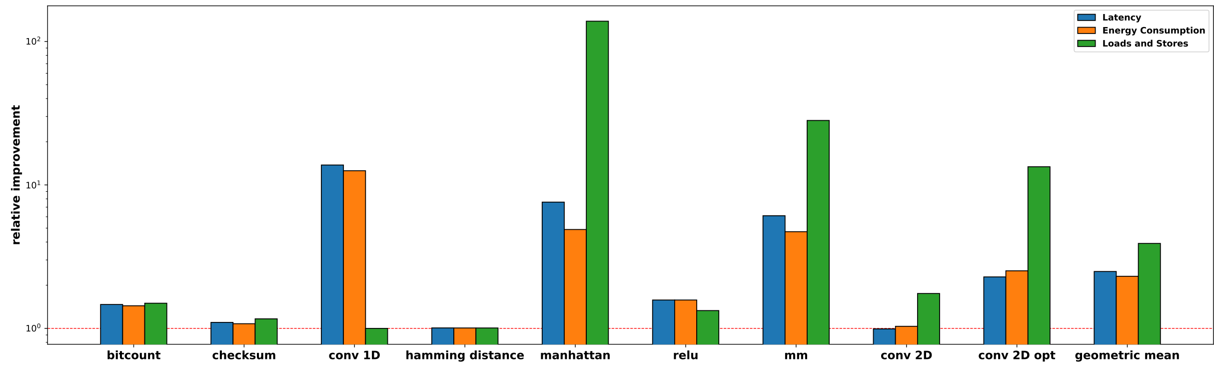

Figure 7.15: The relative improvement in terms of latency, energy consumption and load/stores for all applications evaluated.

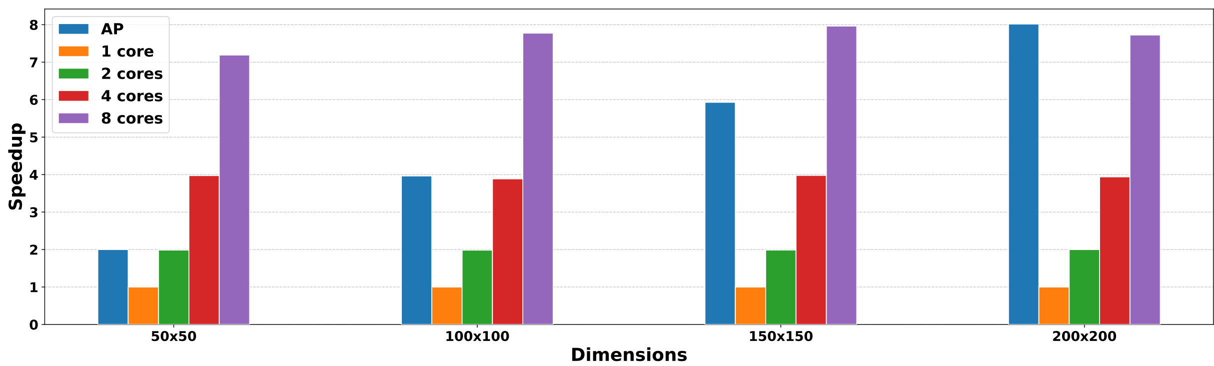

Figure 7.16: Associative Processing matrix multiplication speed-up over a single-core CPU baseline, compared with speed-up for multi-threading with 2, 4, and 8 cores.

the matrices, suggesting that a larger SPM can provide higher speed-ups for larger matrices even in comparison with larger multi-core CPUs. Note that this assessment has a perspective of speed of execution. In a 128 KB AP, both the area and the energy consumed must be taken into account. For this reason, several works are migrating to non-volatile memory technologies that take up less space and spend less energy [32, 34, 52, 56].

Table 7.1: Relative improvements of all tested application kernels in their respective input sizes.

		Relative improvements		
Applications	**Inputs for 16KB AP**	**Latency**	**Energy Consumption**	**load/store operations**
bitcount	1 - 16000 bytes	1.468x	1.438x	1.498x
checksum	1 - 16000 bytes	1.101x	1.078x	1.166x
conv 1D	1 - 4000 bytes	13.77x	12.560x	1x
hamming distance	1 - 16000 bytes	1.008x	1.007x	1.007x
manhattan distance	3 - 2000 bytes per vector	7.590x	4.890x	138.471x
ReLU	1 - 8000 bytes	1.577x	1.577x	1.331x
matrix multiply	3x3 - 72x72 two matrix	5.950x	4.722x	25.894x
2D Convolution	3x3 filter 3x3 - 40x40 image	0.992x	1.034x	1.751x
2D Convolution optimized	3x3 filter 3x3 - 40x40 image	2.285x	2.520x	13.39x
geometric mean	-	**2.485x**	**2.308x**	**3.874x**

Table 7.2: 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

7.3 Simulation performance

Finally, we evaluate the performance of RV-Across itself to run the execution scenarios. Table 7.2 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 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, writes, miswrites, and a full trace for the designer to understand the behavior of the operation. All these bring a significant impact on execution time, but add data to get a more accurate simulation of the associative operations.

Chapter 8

Conclusion

This work presents an exploration of the potential of associative processing using RV-Across, which is a simulator for associative operations, in different application kernels under different scenarios. Applying the models for calculating latency and energy consumption for CPU-only and multi-CPU scenarios versus a model with the Associative Processor. We found a relative gain of 2x for latency, 2x for energy consumption and 3x for load/store operations against the CPU-only model. And against the multi-CPU model, running the multiplication in parallel, the model with the AP can be faster. In addition to the relative gains, the 16 KB Associative Processor CAM would use half the area of an RISCV Ariane core.

On the other hand, to extract parallelism from the Associative Processor, it is necessary to keep the data properly organized. The organization process, as we saw in the case of 2D convolution, can be costly to the point of compromising the efficiency of the system. Another issue is that to obtain good efficiency, it is necessary to work with the smallest possible word size. This is because the performance of the operation is guided by the word size. Energy consumption is mainly affected by the increase in word size for two reasons: increased area and impact of run-time on high power hardware. The word size issue is linked to floating point operations that are poorly explained in the literature, which leads to the conclusion that associative processing is more exploited for integers. There are also gaps in the literature in terms of the AP controller overhead used to implement the associative algorithm.

There is a difficulty in finding associative processing simulators, such as RV-Across, which are open and have a defined model for assessing latency and energy consumption (*https://github.com/JEvSilv/riscv-isa-sim*). Some researchers implement the simulators and present them in the evaluation description as an in-house simulator, thus hindering validation and comparison. RV-Across is an open simulator that provides apparatus for evaluating both the behavior of the associative operation and the performance in an architectural model. Our simulator can be customized both in terms of parameters and in the way of processing. RV-Across was described in the article RV-Across: An Associative Processing Simulator [25] published in the 2020 High Performance Computer Systems Symposium (WSCAD).

As future work we intend to improve our models and make comparisons with GPU, Tensor Processor Unit (TPU) and other PIM approaches. In the short term we intend

to improve the performance of RV-Across and provide support for an independent execution of the Associative Processor. We also want to explore floating point operations on the Associative Processor and implement an AP in FPGA to obtain more accurate results.

Bibliography

[1] J. Ahn, S. Yoo, O. Mutlu, and K. Choi. Pim-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 336–348, June 2015.

[2] Krste Asanovi Andrew Waterman and SiFive Inc. The RISC-V Instruction Set Manual. https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf, 2011. Accessed: 2019-07-31.

[3] Shaahin Angizi, Jiao Sun, Wei Zhang, and Deliang Fan. AlignS: A processing-in-memory accelerator for dna short read alignment leveraging sot-mram. In *DAC*, pages 144:1–144:6, 2019.

[4] Shaahin Angizi, Jiao Sun, Wei Zhang, and Deliang Fan. Pim-aligner: A processing-in-mram platform for biological sequence alignment. *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2020.

[5] I. Arsovski, T. Hebig, D. Dobson, and R. Wistort. A 32 nm 0.58-fj/bit/search 1-ghz ternary content addressable memory compiler using silicon-aware early-predict late-correct sensing with embedded deep-trench capacitor noise mitigation. *IEEE Journal of Solid-State Circuits*, 48(4):932–939, 2013.

[6] Amirali Boroumand, Saugata Ghose, Youngsok Kim, Rachata Ausavarungnirun, Eric Shiu, Rahul Thakur, Daehyun Kim, Aki Kuusela, Allan Knies, Parthasarathy Ranganathan, and Onur Mutlu. Google workloads for consumer devices: Mitigating data movement bottlenecks. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, page 316–331, New York, NY, USA, 2018. Association for Computing Machinery.

[7] Chao Cheng and Keshab K Parhi. Fast 2d convolution algorithms for convolutional neural networks. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 67(5):1678–1691, 2020.

[8] Ping Chi, Shuangchen Li, Cong Xu, Tao Zhang, Jishen Zhao, Yongpan Liu, Yu Wang, and Yuan Xie. Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, pages 27–39, Piscataway, NJ, USA, 2016. IEEE Press.

[9] ML Glossary Community. Activation functions. https://ml-cheatsheet.readthedocs.io/en/latest/activation_functions.html#relu, 2018. Accessed: 2021-01-01.

[10] Hybrid Memory Cube Consortium. Hmc. http://hybridmemorycube.org/, 2018. Accessed: 2018-10-20.

[11] Triscend Corporation. What is the delay from dma request to dma acknowledge. prevailing-technology.com, 2002. Accessed: 2018-10-20.

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

[13] T. Daulby, A. Savanth, G. V. Merrett, and A. S. Weddell. Improving the forward progress of transient systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 1–1, 2020.

[14] Jeff Draper, Jacqueline Chame, Mary Hall, Craig Steele, Tim Barrett, Jeff LaCoss, John Granacki, Jaewook Shin, Chun Chen, Chang Woo Kang, Ihn Kim, and Gokhan Daglikoca. The architecture of the diva processing-in-memory chip. In *Proceedings of the 16th International Conference on Supercomputing*, ICS '02, pages 14–25, New York, NY, USA, 2002. ACM.

[15] Steve Farrer. 80186/80188 dma latency. datasheets.chipdb.org/Intel, 1989. Accessed: 2018-10-20.

[16] Mingyu Gao, Jing Pu, Xuan Yang, Mark Horowitz, and Christos Kozyrakis. Tetris: Scalable and efficient neural network acceleration with 3d memory. *SIGOPS Oper. Syst. Rev.*, 51(2):751–764, April 2017.

[17] M. Gokhale, B. Holmes, and K. Iobst. Processing in memory: the terasys massively parallel pim array. *Computer*, 28(4):23–31, April 1995.

[18] Saransh Gupta, Mohsen Imani, Harveen Kaur, and Tajana Simunic Rosing. NNPIM: A Processing In-Memory Architecture for Neural Network Acceleration. *IEEE TC*, 9340(c):1–1, 2019.

[19] M. Imani, S. Gupta, and T. Rosing. Genpim: Generalized processing in-memory to accelerate data intensive applications. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1155–1158, March 2018.

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

[21] M. Imani, A. Rahimi, and T. S. Rosing. Resistive configurable associative memory for approximate computing. In *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1327–1332, 2016.

[22] Mohsen Imani, Saransh Gupta, Yeseong Kim, Minxuan Zhou, and Tajana Rosing. Digitalpim: Digital-based processing in-memory for big data acceleration. In *Proceedings of the 2019 on Great Lakes Symposium on VLSI*, GLSVLSI '19, pages 429–434, New York, NY, USA, 2019. ACM.

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

[24] L. Jiang, M. Kim, W. Wen, and D. Wang. Xnor-pop: A processing-in-memory architecture for binary convolutional neural networks in wide-io2 drams. In *2017 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, pages 1–6, 2017.

[25] João Fabrício Filho Jonathas Silveira, Isaias Felzmann and Lucas Wanner. Rv-across: An associative processing simulator. RV-Across full paper link, 2020. Accessed: 2018-10-20.

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

[27] K. Kaur. A survey on internet of things – architecture, applications, and future trends. In *2018 First International Conference on Secure Cyber Computing and Communication (ICSCCC)*, pages 581–583, Dec 2018.

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

[29] Jeremie S. Kim, Damla Senol Cali, Hongyi Xin, Donghyuk Lee, Saugata Ghose, Mohammed Alser, Hasan Hassan, Oguz Ergin, Can Alkan, and Onur Mutlu. GRIM-Filter: Fast seed location filtering in DNA read mapping using processing-in-memory technologies. *BMC Genomics*, 19(2):89, 2018.

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

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

[32] H. Li, H. Jin, L. Zheng, and X. Liao. Resqm: Accelerating database operations using reram-based content addressable memory. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(11):4030–4041, 2020.

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

[34] Sparsh Mittal. A survey of reram-based architectures for processing-in-memory and neural networks. *Machine learning and knowledge extraction*, 1(1):75–114, 2019.

[35] Onur Mutlu, Saugata Ghose, Juan Gómez-Luna, and Rachata Ausavarungnirun. Enabling practical processing in and near memory for data-intensive computing. In *Proceedings of the 56th Annual Design Automation Conference 2019*, DAC '19, pages 21:1–21:4, New York, NY, USA, 2019. ACM.

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

[37] Hoang Anh Du Nguyen, Jintao Yu, Muath Abu Lebdeh, Mottaqiallah Taouil, Said Hamdioui, and Francky Catthoor. A classification of memory-centric computing. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 16(2):1–26, 2020.

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

[39] Dhinakaran Pandiyan and Carole-Jean Wu. Quantifying the energy cost of data movement for emerging smart phone workloads on mobile platforms. In *2014 IEEE International Symposium on Workload Characterization (IISWC)*, pages 171–180. IEEE, 2014.

[40] João Paulo and Cardoso De Lima. PIM-gem5 : a system simulator for Processing-in-Memory design space exploration. Master's thesis, Universidade Federal do Rio Grande do Sul, 2019.

[41] Bruno E. Forlin Paulo C. Santos and Luigi Carro. *sim*^{2} pim: A fast method for simulating host independent & pim agnostic designs. In *2021 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2021.

[42] Zaid Qureshi, Vikram Sharma Mailthody, Seung Won Min, I Chung, Jinjun Xiong, Wen-mei Hwu, et al. Tearing down the memory wall. *arXiv preprint arXiv:2008.10169*, 2020.

[43] J. Saikia, S. Yin, Z. Jiang, M. Seok, and J. Seo. K-nearest neighbor hardware accelerator using in-memory computing sram. In *2019 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, pages 1–6, 2019.

[44] Paulo Cesar Santos, João Paulo C. de Lima, Rafael F. de Moura, Hameeza Ahmed, Marco A. Z. Alves, Antonio C. S. Beck, and Luigi Carro. Exploring IoT Platform with Technologically Agnostic Processing-in-memory Framework. In *INTESA*, pages 1–6, 2018.

[45] Vivek Seshadri, Donghyuk Lee, Thomas Mullins, Hasan Hassan, Amirali Boroumand, Jeremie Kim, Michael A. Kozuch, Onur Mutlu, Phillip B. Gibbons, and Todd C. Mowry. Ambit: In-memory accelerator for bulk bitwise operations using commodity dram technology. In *Proceedings of the 50th Annual IEEE/ACM*

International Symposium on Microarchitecture, MICRO-50 '17, pages 273–287, New York, NY, USA, 2017. ACM.

[46] Ali Shafiee, Anirban Nag, Naveen Muralimanohar, Rajeev Balasubramonian, John Paul Strachan, Miao Hu, R. Stanley Williams, and Vivek Srikumar. Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars. *SIGARCH Comput. Archit. News*, 44(3):14–26, June 2016.

[47] Gurusiddayya Hiremath Shashi Kumar. Low power implementation of risc-v processor. *IOSR Journal of VLSI and Signal Processing (IOSR-JVSP)*, 6(3):59, 2016.

[48] Robert A Van De Geijn and Jerrell Watts. Summa: Scalable universal matrix multiplication algorithm. *Concurrency: Practice and Experience*, 9(4):255–274, 1997.

[49] Bing Xu, Naiyan Wang, Tianqi Chen, and Mu Li. Empirical evaluation of rectified activations in convolutional network. *arXiv preprint arXiv:1505.00853*, 2015.

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

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

[52] Hasan Erdem Yantir, Ahmed M. Eltawil, and Fadi J. Kurdahi. Approximate memristive in-memory computing. *ACM Trans. Embed. Comput. Syst.*, 16(5s):129:1–129:18, September 2017.

[53] Hasan Erdem Yantir, Ahmed M. Eltawil, and Fadi J. Kurdahi. Approximate memristive in-memory computing. *ACM Trans. Embed. Comput. Syst.*, 16(5s):129:1–129:18, September 2017.

[54] L. Yavits, S. Kvatinsky, A. Morad, and R. Ginosar. Resistive associative processor. *IEEE CAL*, 14(2):148–151, 2015.

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

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

[57] Leonid Yavits, Roman Kaplan, and Ran Ginosar. PRINS: resistive CAM processing in storage. *CoRR*, abs/1805.09612, 2018.

[58] F. Zaruba and L. Benini. The cost of application-class processing: Energy and performance analysis of a linux-ready 1.7-ghz 64-bit risc-v core in 22-nm fdsoi technology. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27(11):2629–2640, 2019.

[59] Dongping Zhang, Nuwan Jayasena, Alexander Lyashevsky, Joseph L. Greathouse, Lifan Xu, and Michael Ignatowski. TOP-PIM: Throughput-oriented programmable processing in memory. In *HPDC*, pages 85–98, 2014.

[60] M. Zhang, Y. Zhuo, C. Wang, M. Gao, Y. Wu, K. Chen, C. Kozyrakis, and X. Qian. GraphP: Reducing communication for pim-based graph processing with efficient data partition. In *HPCA*, pages 544–557, 2018.