A Survey on Versatile Embedded Machine Learning Hardware Acceleration

Pierre Garreau, Pascal Cotret, Julien Francq, Jean-Christophe Cexus, Loïc Lagadec

A Survey on Versatile Embedded Machine Learning Hardware Acceleration (pre-print version)

Pierre Garreau[1,2], Pascal Cotret[2], Julien Francq[3], Jean-Christophe Cexus[2], and Loïc Lagadec[2]

[1]Chair of Naval Cyber Defense, Ecole navale, France
[2]Lab-STICC, UMR CNRS 6285, ENSTA, France
[3]CERT, Naval Group, France

Abstract

This survey investigates recent developments in versatile embedded Machine Learning (ML) hardware acceleration. Various architectural approaches for efficient implementation of ML algorithms on resource-constrained devices are analyzed, focusing on three key aspects: performance optimization, embedded system considerations (throughput, latency, energy efficiency) and multi-application support. The survey then explores different hardware acceleration strategies, from custom RISC-V instructions to specialized Processing Elements (PEs), Processing-in-Memory (PiM) architectures and co-design approaches. Notable innovations include flexible bit-precision support, reconfigurable PEs, and optimal memory management techniques for reducing weights and (hyper)-parameters movements overhead. Subsequently, these architectures are evaluated based on the aforementioned key aspects. Our analysis shows that relevant and robust embedded ML acceleration requires careful consideration of the trade-offs between computational capability, power consumption, and architecture flexibility, depending on the application.

1 Introduction

Machine Learning (ML) is a subset of Predictive Artificial Intelligence (AI), with techniques such as Deep Neural Networks (DNNs) in order to forecast outcomes. ML algorithms fall into three primary categories:

- Supervised learning, where models are trained on labeled data to predict outputs;
- Unsupervised learning, where patterns are found in unlabeled data;
- Semi-supervised learning, which is in between the aforementioned categories, leveraging both labeled and unlabeled data.

Traditionally, both training and inference phases used to take place on servers with huge computational power and low energy consumption restrictions. Thus, in edge computing systems, it can be tempting for the edge(s) to (securely) delegate heavy computations to the server. Even theoretically valid, this approach is sometimes impractical. Let us take an example of an Unmanned Aerial Vehicle (UAV) benefiting from an ML-based Intrusion Detection

System (IDS) capacity delegated to a server. The UAV would then have to continuously send data to the distant server that will compute the inference of the model. In this scenario, the UAV will use a significant amount of power for data transmission; therefore, its operating life will be drastically reduced. Moreover, there is a high risk of packet loss. Finally, real-time intrusion detection is needed, so the induced communication latencies are undesirable.

Embedding ML inside the edge would then be the appropriate solution in such cases. However, embedded devices have limited power resources. Another basic observation is that ML algorithms and the workload they induce are inefficiently processed on general-purpose Central Processing Units (CPUs) [16]. Actually, this is due to the objects handled when dealing with ML algorithms (for instance, vectors, matrices or arrays of parameters). Figure 1 describes the computations behind the process of a fully connected layer in a DNN.

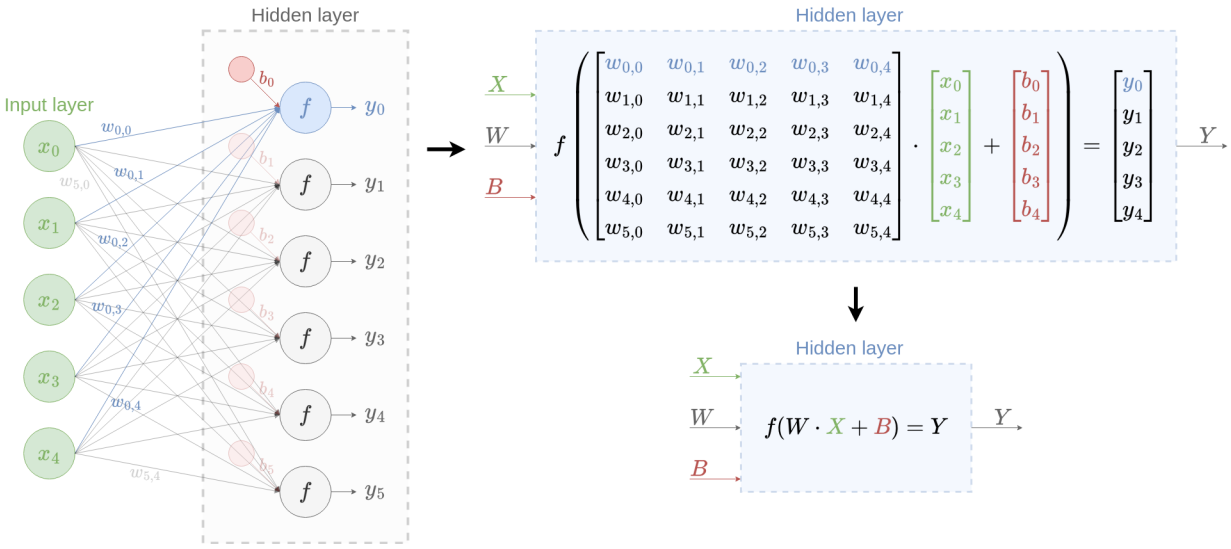


Figure 1: Example of computations behind the processing of a hidden layer in a DNN.

Handled objects are matrices with custom sizes, depending on the processed layer. Multiplications, additions and non-linear function evaluation (such as f in Figure 1) are performed for that category of layer. Traditional CPUs are not optimized to perform computations on such objects, like GEneral Matrix Multiplication (GEMM) or GEneral Matrix-Vector multiplication (GEMV). Creating custom hardware architectures appears as an interesting solution to efficiently execute ML algorithms. This work explores the main approaches proposed in related works to accelerate such algorithms.

In addition, embedded devices may benefit from ML for several tasks. For instance, a UAV could use ML for object detection, as well as for environmental data processing or IDS. In this context, building versatile architectures could be relevant. Nonetheless, the significant diversity among ML algorithms is problematic for that purpose. Basically, elementary operations behind DNNs are GEMM or GEMV, which are both built around accumulations, also known as Multiply and ACcumulate (MAC) operations, but also *pooling*, an operation that reduces the spatial dimensions feature maps, or *activation functions*, i.e. non-linear functions allowing Convolutional Neural Networks (CNNs) (or DNNs) to learn and model complex patterns. On the other hand, elementary operations behind Random Forests (RFs) are comparisons, therefore requiring different hardware. This diversity indicates that the developed hardware architectures are designed with considerations for reconfigurability, scalability, adaptability, and other similar factors.

Recently, numerous studies have aimed at summarizing the various architectures and strategies developed to make hardware more suitable for the deployment of Neural Networks (NNs) inside edges. Zaman et al. [41], as well as Mohaidat et al. [25], reviewed several Deep

Learning (DL) hardware accelerators going beyond data routing, specialized Processing Elements (PEs) and on-chip efficient memories. Silvano et al. [30] also devoted a section of their survey to the challenges of embedded hardware acceleration. However, in this work, we aim at identifying the challenges and objectives of hardware acceleration of ML algorithms on embedded devices, while considering multi-model applications, which is overlooked in the aforementioned studies and will serve as the main point of this survey. Then, this work aims to provide an extensive vision of methods, tools and approaches that have been leveraged over the past few years for versatile ML hardware acceleration on embedded devices. For that matter, architectures that have been recently developed will be deeply presented. An evaluation of these architectures will also be proposed to understand how a developer can benefit from them.

Remaining sections are organized as follows. Section 2 presents the challenges and objectives of versatile embedded ML hardware acceleration. Section 3 deeply elaborates the architectures and the features developed over the past few years. Section 4 discusses the features found in the previous section, before concluding.

2 Challenges and objectives

In recent years, efforts in hardware acceleration for ML algorithms have focused on optimizing the hardware performance [41, 25, 28], while ensuring the correctness of the inference output. More specifically, they have evaluated performance criteria such as throughput, latency, and processing speed to assess the actual efficiency of acceleration solutions. In parallel, standard ML metrics have been used to verify that these optimizations do not compromise the overall performance of the algorithms themselves [25]. When applied to embedded systems, performance of such architectures cannot be limited to these metrics. That is why, resources consumption and area efficiency arise as unavoidable variables to quantify the performance of any embedded hardware acceleration architecture [29]. On top of that, considering the versatile aspect of ML algorithms, adding challenges such as adaptability to different applications and scalability of the hardware might be convenient to embed a large panel of ML algorithms in an optimal manner.

2.1 Performance evaluation

The main purpose of ML hardware acceleration is to alleviate the deficiency of performance in traditional general-purpose hardware architectures. Standard CPUs are not meant to perform data intensive tasks such as ML algorithms inference. As stated by Zaman et al. [41], due to the Von Neumann architecture, CPU performances are bounded by cache size and external memory instruction fetching. Hence, their architecture can be enhanced, and is therefore the main objective of hardware acceleration. The most common variables that are being brought up are throughput and latency: these two correlated variables highlight the capability of the hardware architecture to accommodate the ML computational workload.

First, throughput is the amount of operations per second the design can achieve and is usually evaluated in *Giga* or *Tera Operations Per Second* (GOPS or TOPS) in the literature. Then, when applied to ML algorithms, latency is the duration needed to compute a single inference of the model that is being processed. Depending on the application, this metric may determine if an algorithm should be chosen or not. For instance, when using an ML algorithm for missile detection on images collected by UAVs, if image processing takes longer than the missile takes to reach its target, embedding such an algorithm would be pointless. Maximizing throughput and minimizing latency are the main challenges to address when designing such architectures. These challenges are technically reflected in various ways [29]. For throughput, it includes the optimization of data mobility, and the use of specific hardware for standard ML

functions (such as GEMM, convolutions, activation functions, etc.). Multi-core architectures arise as powerful solutions to enhance parallelism and further improve throughput. For latency, it includes the exploration of CPU-accelerator configurations that could improve interactions between these two entities or the use of on-chip buffers to shorten the distance traveled by the data from memory to processing elements. On top of that, some architectures [32, 6] have recently leveraged Processing-in-Memory (PiM) elements to overcome this travel distance, moving the processing units directly into the memory.

When dealing with ML algorithms, performance also entails guaranteeing the invariability of the ML algorithm, thereby preserving standard ML metrics such as accuracy or precision for instance. This is important to mention as some hardware acceleration processes, such as quantization [37, 1], described later in this survey, involve altering parameters and weights of the ML algorithm itself, for energy efficiency purpose mainly. Hence, designers need to ensure their optimization paradigms will not impair algorithms capability to fulfill their tasks. For example, some architectures [11] compute the hardware inference accuracy, corresponding to the standard inference accuracy but with the potentially quantized and optimized model that is loaded on the hardware, and compare it to the standard accuracy computed by traditional ML libraries like PyTorch.

Depending on the application the architecture is intended to be used for, performance can be related to area efficiency [25], corresponding to the amount of hardware resources needed to perform computations. Indeed, two architectures with the same throughput and latency, but one using half the hardware resources compared to the other, do not exhibit the same performance due to differences in area efficiency. This metric can be challenging to quantify, as, depending on the hardware architecture and the involved ML paradigms, some models might be more complex to process than others. This is likely due to the large range of different ML algorithms that can be implemented. Technically, area efficiency can be enhanced by the layout or the configuration of the hardware itself. Some architectures, especially some Instructions Set Architectures (ISAs) like RISC-V [14], can be optimized to be more suitable to ML workload [29], improving the efficiency of the hardware resources, therefore improving the area efficiency.

2.2 Embedded system considerations

Applied to embedded systems such as UAVs, hardware acceleration performance evaluation is not restricted to the metrics mentioned in Section 2.1, and must include energy efficiency evaluation. For this reason, Graphics Processing Unit (GPU)-based acceleration of ML algorithms is usually not a viable solution in this context [13].

First, embedded systems carry power consumption constraints, as they are powered by batteries. Most of the time, only a small portion of the power available is dedicated to intelligence, with most of it reserved for the system primary functions. For instance, Valente et al. [35] pointed out that only 5 to 15% of the total power envelope of nano-UAVs is dedicated to computation, leaving the remaining portion for vital functions such as flying or communication. Thus, designing an architecture with the maximum throughput and the minimum latency is not enough for these systems. However, considering variables such as throughput per Watt, which is evaluated in GOPS/W or TOPS/W, is significantly more appropriate. There are numerous technical and hardware solutions that have been developed for these optimization problems. Basically, researchers identified the main reasons for the ML workload to be energy-consuming on traditional architectures.

Then, most of the ML calculations includes operations on matrices and vectors, called GEMM or GEMV, that general-purpose CPUs fail to process efficiently. As a result, among the features detailed in Section 3, designers have mainly focused on developing dedicated hardware [34, 26, 5], and Single-Instruction-Multiple-Data (SIMD) instructions and parallelism [40]

to address this challenge. These solutions often require more hardware than standard architecture, but does not imply increased power consumption, because the hardware is designed and optimized for the ML computations and is therefore more efficient than common architectures. Furthermore, specific hardware has also been developed to address the complexity of ML calculations, including convolutions, activation functions and other specific operations required by common DNN models. For that matter, various techniques have been employed. For instance, both AMD/Xilinx and Intel address this challenge, but in two completely different ways [34]. AMD developed specific AI Engines (AIEs) and organized these AIEs in 2D arrays to enhance parallelism, while Intel revised standard Digital Signal Processing (DSP) blocks to further improve computations. Actually, leveraging hardware efficiency thanks to these methods is an attempt to improve the so-called "active area", metric that Bavikadi et al. [6] highlighted and tried to increase when designing their ML accelerator.

Furthermore, another reason related to the energy consumption of ML algorithms on general-purpose architectures is the number of weights and parameters required, as well as the precision of the computations. For example, common DNNs include millions of parameters, while Large Language Models (LLMs) require more than a trillion of parameters. Handling this amount of data on an embedded device would be, if it is possible, energy-intensive. On top of that, certain parameters in ML algorithms, particularly in DNNs, may not be essential for the algorithm to operate properly. Hardware acceleration seeks to optimize the trade-off between the ML model performance, system hardware performance (considering the metrics previously discussed), and energy consumption. For instance, reducing the number of neurons in a DNN layer may result in a minimal loss of accuracy compared to the original DNN, while significantly increasing throughput, reducing latency, and lowering energy consumption. Hence, researchers have explored methods to achieve such trade-offs. Among other methods described later, pruning and quantization appear as extremely powerful. First, even if pruning is more likely to be a software optimization, it has a hardware impact as it allows finding and deleting, in DNNs, neurons that have a negligible impact on the outputs, thus changing the shape of the model and therefore impacting its hardware implementation. Then, quantization is focusing on reducing the cost of each computation by using cheaper representations of the parameters involved in ML computations. For instance, instead of using 64 bits for the representation of each weight of DNN and using floats, representing these weights using 8 or 16 bits and integers might not have a significant impact on accuracy but will considerably reduce the power consumption of the hardware architecture [37, 20].

Eventually, the last main reason for the general-purpose architectures not to be energy-efficient when working with ML workload is linked to the amount of parameters as well. As mentioned earlier, these algorithms require a lot of parameters. However, in traditional architectures, the size of the memory elements near the PEs is too modest to store all the parameters of such algorithms, leading to a lot of data movements (primarily loads and stores) between distant memory and processing units. These interactions have been identified as a bottleneck by Mohaidat et al. [25] and, according to them, parameters related to ML workload must be stored in on-chip memory. To address this challenge, solutions consist in adding memory buffers to the optimized hardware designed for computations, directly connecting the distant memory to the processing units, or using new elements such as PiM units [6, 42].

That being said, it is of paramount importance to mention that the hardware optimization and acceleration of ML workload is strongly related to the algorithm that will be implemented on the architecture. Most of the time, hardware accelerators are expected to be fully optimized on given algorithms, or even only on specific instantiations (models). For instance, Liang et al. [22] developed an architecture with the intent to be efficient on LeNet-5, a specific CNN instance. To tackle this issue, some efforts have focused on generating specific hardware, given a ML model [11, 43]. The strategy behind these frameworks or generators begins with a prediction

step, through which the proposed tool is able to efficiently and reliably predict the performances of hardware architectures by computing all the variables mentioned earlier (*e.g.* throughput, power consumption, area usage, energy-efficiency, etc.). Based on the metrics computed during this first step, hardware can automatically be generated and further optimized by exploring the design space. Actually, Han et al. [11] tried to extend this strategy by adding a retraining step, by injecting quantization, variation and non-linearity so that it adapts itself to the hardware non-idealities, for the ML model to be even more efficient on the generated hardware. They also conveyed the idea of trade-offs that has been previously discussed by letting the designers choose whether the generated hardware is more focused on energy efficiency, on hardware performance or on the accuracy of the ML model itself.

2.3 Multi-application

Given the vast array of ML models that have been developed over the years, ranging from linear regression to NNs, as well as RFs or Support Vector Machines (SVMs), the development of flexible and reconfigurable architectures has become inevitable. In addition, these models can complete a wide variety of tasks, from object detection or classification to speech recognition, as well as image, video or text generation. That multi-purpose nature introduces new challenges and objectives for hardware acceleration. First, reconfigurability is often achieved through the use of Field Programmable Gate Arrays (FPGAs), allowing the designers to perpetually customize the hardware, reconfiguring it anytime they need to change the model type or size. Since performance and energy efficiency are closely tied to the embedded ML model, these integrated circuits are ideally suited to meet the demands of ML workload processing, and frameworks discussed in the previous paragraph further facilitate reconfigurability. Furthermore, some architectures, such as the one described by Bavikadi et al. [6], provide configurable hardware, usually PEs with different operating modes, that can be more efficient on a wider range of models than standard hardware. Eventually, Khalil et al. [18] developed a hardware architecture based on Network-on-Chip (NoC) paradigm. This method, consisting of routers and PEs smartly connected, is an uncommon way to address this reconfigurability challenge. By choosing the number of computing "nodes" inside PEs, the architecture adapts itself to the ML model, enhancing performance.

Then, as mentioned by Mohaidat et al. [25], in order to assess the flexibility of hardware acceleration architectures, diverse workloads are necessary, meaning various model natures, sizes and complexities. If the designed hardware architecture accelerates decently only a small part of ML algorithms, it will obviously not be meant for multi-purpose applications. To enhance flexibility, it is essential to carefully examine the specific characteristics of all ML algorithms. First, as stated earlier, quantization of weights and parameters can be relevant in certain circumstances for energy efficiency and performance purposes. Thus, from a model to another, or even from one layer to another in a NN for instance, numbers representation can be different, ranging from 1 or 2-bit integers, to 32 or 64-bit floating-point or fixed-point floats.

However, handling all these data types can be challenging at the hardware level. That ability is often referred as multi-precision and is usually sought after. For that matter, configurable hardware discussed in the previous paragraph is relevant. For instance, Bavikadi et al. [6] designed special Look-Up Tables (LUTs), working on several modes for the hardware to be efficient on several data types, thus further enhancing the flexibility of the architecture. Various works [42, 15] implemented arrays of units that can be cleverly configured depending on the weights and parameters type. This way, not only the hardware is able to optimize the ML model computations, but it is also fully flexible and can be valuable when dealing with various ML model types.

3 Custom hardware architectures

Over the past few years, designers have considered various strategies to integrate ML algorithms in embedded systems. These strategies and more specifically the hardware architectures derived from them, can be summarized according to the high-level diagram presented in Figure 2. In this figure, as well as in all figures extracted from existing works, features are represented in blue for host and off-chip hardware, red for on-chip memory solutions, and green for dedicated specific hardware. First, Section 3.1 presents architectures that have focused on including specific hardware to process standard ML functions. Numerous works, presented Section 3.2, have leveraged custom ISA to enhance specific tasks on hardware architectures and increase their efficiency. Although hardware acceleration has benefited from focusing on computations acceleration, data transfers required by these types of workload paved the way for PiM architectures and data movements optimizations. Section 3.3 details architectures that leverage PiM-like technologies. Eventually, co-design approaches recently emerged as powerful strategies to generate efficient hardware for such workload, as hardware mainly depends on the application and the model the architecture is supposed to accelerate. These approaches are described in Section 3.4. All the architectures developed in this section are gathered and categorized in Figure 3. The categorization seeks to sort architectures based on the main challenge they address or the key feature they leverage.

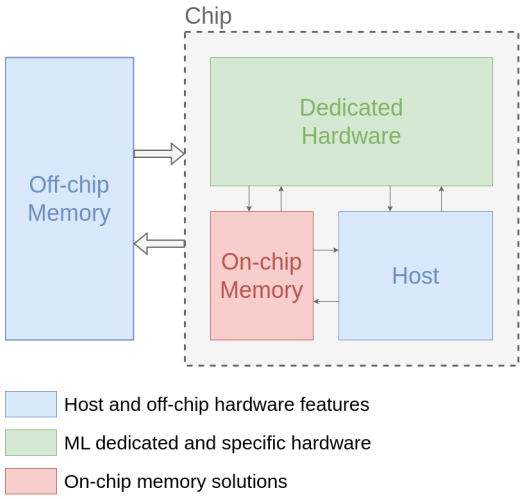


Figure 2: High-level diagram of the hardware architectures designed for ML acceleration.

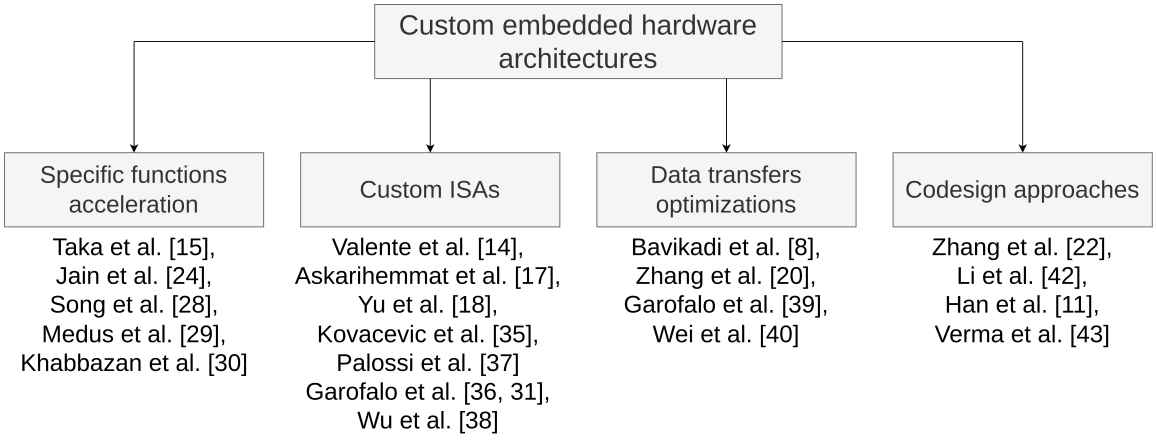
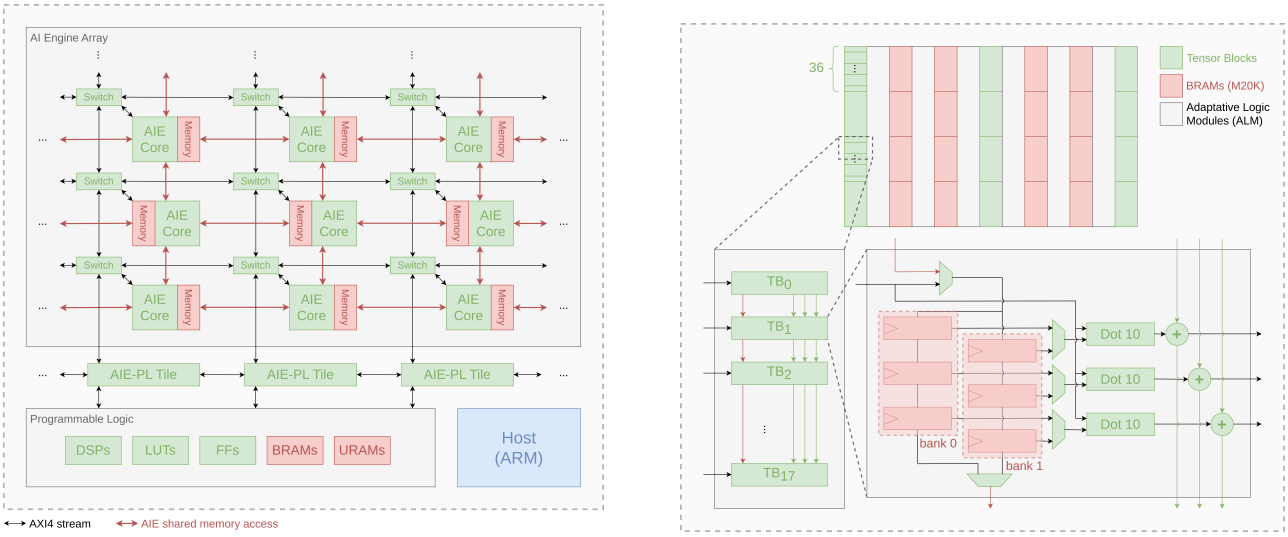


Figure 3: Classification of the architectures described in Section 3.

3.1 Standard ML functions acceleration

Firstly, to overcome some challenges discussed in Section 2, including special hardware for specific ML functions seems appropriate. Intrinsically, ML algorithms induce heavy workload and engage data types like vectors or matrices that general purpose micro-architectures fail to efficiently process. Therefore, offloading ML workload to hardware specifically designed for such computations has become a critical consideration for designers in recent years. This concept can be technically interpreted in various manners.

For instance, Taka et al. [34] present strategies to implement GEMM acceleration on two different AI-optimized FPGA state-of-art architectures: AMD/Xilinx Versal ACAP and Intel Stratix 10 NX. These architectures are presented in Figure 4 and aim at optimizing GEMM operations, as a lot of ML (specifically DL) workload relies on these operations.



(a) Versal ACAP architecture.

(b) Architecture of Stratix 10 NX Tensor Blocks.

Figure 4: AMD/Xilinx vs. Intel AI-optimized FPGAs architectures [34].

On the first hand, the Versal ACAP architecture includes a 2D array of AIE tiles, each tile consisting of an AIE core, a memory unit and an interconnect. AIE cores [2] are the processing units responsible for performing GEMM operations as they feature a Very Long Instruction Word (VLIW) programmable processor with SIMD units. Up to 400 AIE cores can be concurrently implemented. Communication between this array and the traditional FPGA resources (Block RAMs (BRAMs), Ultra RAMs (URAMs), DSPs, LUTs, etc.) in the Programmable Logic (PL) is ensured by a row of AIE-PL tiles, providing AXI4-Streaming interface. This architecture is controlled by ARM processors. Eventually, tools such as Vitis HLS [3] are made available for programming the AIE cores and the PL. On the other hand, Stratix 10 NX architecture replaced variable-precision DSP blocks with Tensor Block (TB), and many of the high-precision and legacy modes are replaced with scalar, vector and tensor operating modes for several data types as well. These TBs perform a dot-product operation, which is a multiplication of vectors resulting in a scalar value − as well as additions for accumulation on partial products thanks to banks of buffers − and fixed-point adders. By cascading these TBs, this architecture is able to propagate and accumulate products. The length of these groups of TBs can be configured to adapt to any ML workload. However, there is no support for programming these TBs using high-level tools; it has to be done in Register-Transfer Level (RTL). Then, Taka et al. [34] leverage the MaxEVA framework [33] to efficiently implement GEMM on Versal ACAP, mainly thanks to memory optimization strategy (on-chip buffer data reuse maximization) and efficient matrix multiplication configuration. Same paradigms are leveraged on Stratix 10 NX architecture, but, as the architecture is significantly different, it is technically translated differently: TBs layout (length of TBs arrays, sets of arrays working in parallel, etc.), the dataflow (synchronization of TBs, operations performed by each TB, etc.), and the memory architecture are optimized. These optimizations depend on the size of the GEMM operations it is supposed to perform, and the generated hardware is meant to be reconfigured accordingly.

Furthermore, Jain et al. [15] also focus on specialized hardware optimizations that target common computational patterns in ML. An overview of the whole architecture is given in Figure 5.

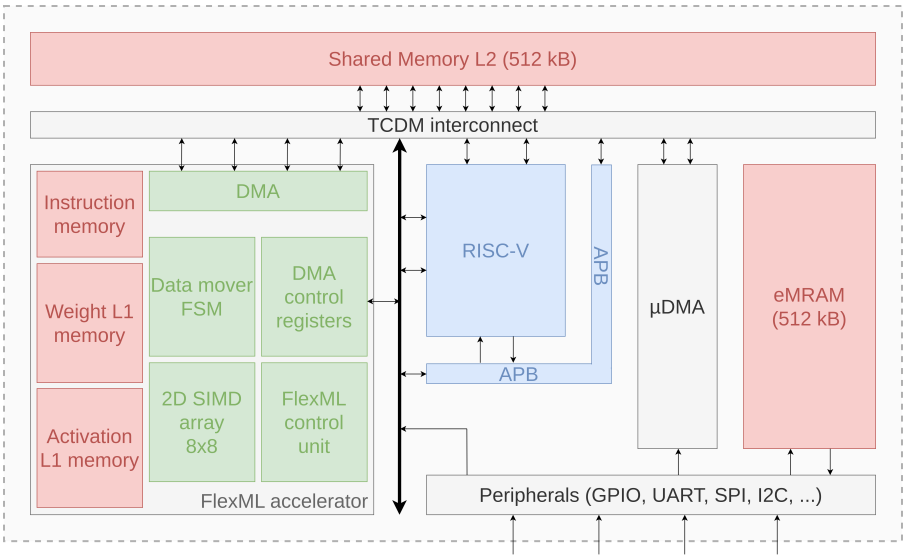
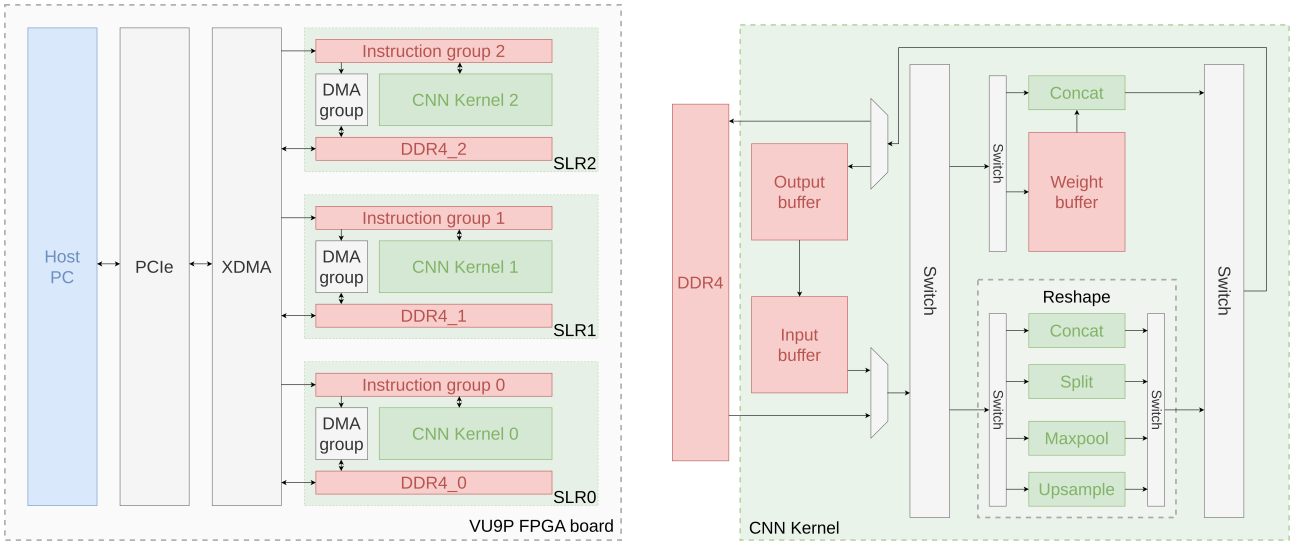


Figure 5: Overview of TinyVers SoC [15].

At the core, the FlexML accelerator developed in this work includes an 8x8 SIMD array of PEs, each containing precision-scalable MAC units that can perform 1/2/4 MAC operations per cycle depending on the selected precision (respectively 8-/4-/2-bit integers). This parallel processing architecture is complemented by an efficient memory hierarchy, where L1 memories for weights and activations operate in a ping-pong manner to overlap data movement with computation. The system achieves further acceleration through runtime dataflow reconfiguration, allowing efficient mapping of both GEMM and GEMV with zero latency overhead. This ability is convenient for multi-purpose applications, allowing to efficiently process a large variety of ML workload. On top of that, additional hardware components such as the non-linear function generator for activation functions and dedicated support for SVM operations incorporate even more diversity in the range of ML workload.

Then, Song et al. [31] designed an architecture that implements three CNN accelerators, called "CNN kernels", on a VU9P chip, each accelerator being constrained to an independent Super Logic Region (SLR) to prevent resource interference. The design, presented in Figure 6, incorporates on-chip buffers using URAM to store intermediate results rather than writing back to Dynamic Random Access Memory (DRAM). By reducing these memory accesses, the architecture's energy efficiency and power consumption is enhanced. As depicted in Figure 6b, in order to offload the computationally intensive workload, the architecture includes specialized operators for commonly used functions in ML like: Conv2d (2D convolution, used in CNNs), Concat (concatenation), Maxpool (pooling operation based on max function, used in CNNs to reduce dimensionality), and Upsample (used to increase the dimensionality of feature maps) functions, all designed with parametric flexibility to accommodate various neural network dimensions.

In addition, the system employs 8-bit quantization for both weights and feature maps, enabling twice the throughput per DSP block. The 8-bit quantization process involves converting 32-bit floating-point format parameters obtained after network training into 8-bit format. Actually, according to Song et al. [31], studies have shown this maintains accuracy loss within 1% while providing significant hardware performance benefits (high throughput and low latency) compared to lower bit widths like 4 or 6 bits. To achieve this, the design leverages both software (TVM compiler, a framework designed for ML accelerators) and hardware (sub-modules



(a) CNN accelerator architecture.

(b) CNN kernel architecture.

Figure 6: Multidie design presented by Song et al. [31]

in the accelerator) features. Consequently, Song et al. [31] presented a powerful architecture for CNN acceleration. Although this architecture may not process other types of ML algorithms efficiently, it adapts to a large range of CNNs and could be employed in multi-purpose applications that require multi-sized CNN models.

Medus et al. [24] created an architecture that employs Neural Processing Elements (NPEs) to achieve standard ML function acceleration. These NPEs contain an Arithmetic Logic Unit (ALU), local BRAM and a scratchpad register (a fast access register for temporary storage). NPEs are arranged in a systolic ring configuration for efficient parallel processing. Each NPE handles weight storage and computations for neural units, while a single Activation Function Block (AFB) serves the entire network.

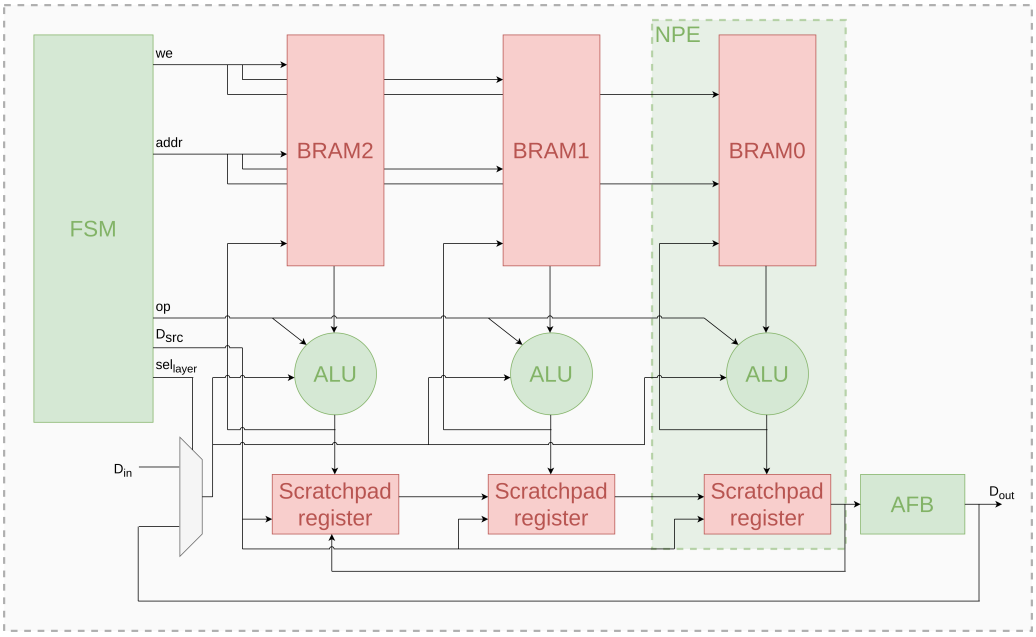
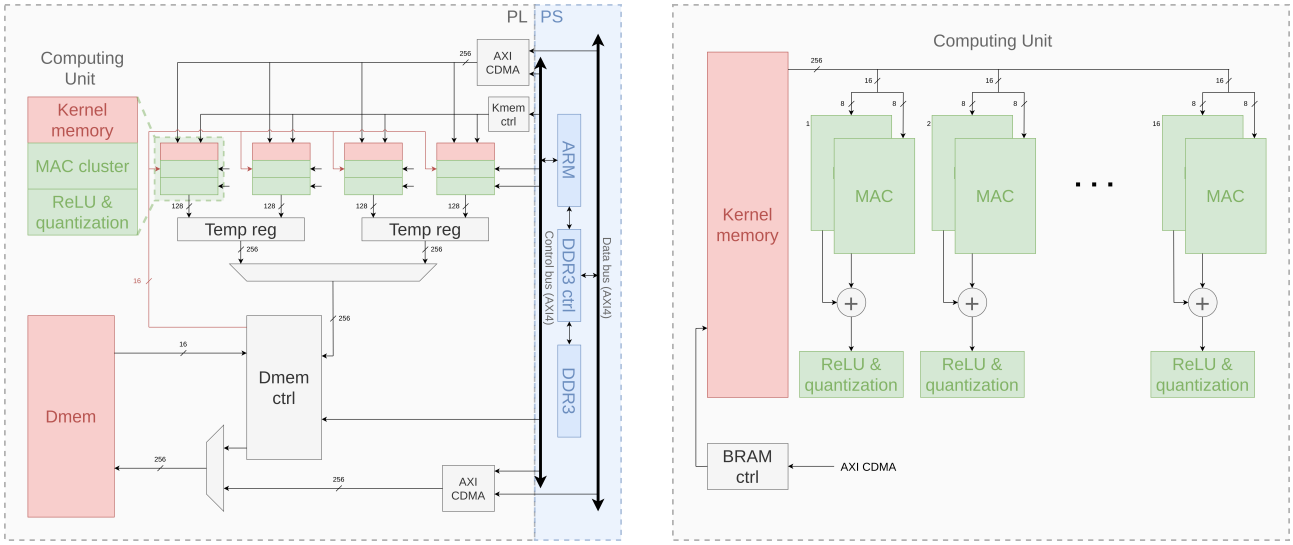


Figure 7: Example of the hardware architecture described by Medus et al. [24]

By using only one AFB for the whole architecture, regardless of the amount of NPEs required, they managed to reduce resource usage compared to traditional approaches. Moreover,

the AFB can be configured to process a few different activation functions, namely Rectified Linear Unit (ReLU), logistic sigmoid and hyberbolic tangent, and, although it is used by every layer, it can be multiplexed to use distinct functions for different layers. The design also achieves acceleration through parallel input processing where all neuron units in a layer process inputs simultaneously, combined with time-multiplexed input handling (one input per clock cycle). This architecture is highly efficient on multi-layer DNNs and scalable as long as hardware resources are available.

Eventually, Khabbazan et al. [17] designed a CNN accelerator that leverages the typical algorithm behind computing a convolution layer, and utilizes loop unrolling and interchange techniques to enhance the computation of these layers on hardware. For that matter, the proposed accelerator architecture, detailed in Figure 8 is divided into three stages: LOAD, CALC, and STORE. During the LOAD stage, input feature maps and kernel weights are stored in on-chip memory using a ping-pong strategy to reduce off-chip memory access. In the CALC stage, computations are handled by parallelized Computing Units (CUs) described in Figure 8b that execute MAC functions using loop unrolling. Each CU comprises MAC arrays and dedicated activation function blocks, allowing 64 outputs to be generated concurrently. Finally, in the STORE stage, CU outputs are temporarily held in registers, then written back to on-chip memory while simultaneously reading new input features, allowing uninterrupted processing. This design optimizes resource usage, minimizes latency, and supports 8-bit quantization, maintaining performance with low-power consumption.



(a) Convolutional accelerator architecture.

(b) Computing Unit (CU) architecture.

Figure 8: Accelerator and special units architecture proposed by Khabbazan et al. [17]

While these architectures can be configured to optimize various ML models, some works have focused on building highly efficient hardware for a specific model. For instance, in order to optimize LeNet-5 network, Liang et al. [22] designed a hardware accelerator, detailed in Figure 9, that leverages specific hardware for convolution, activation, pooling and fully connected operations.

Then, this hardware is utilized in 2 different ways, leading to 2 distinct accelerators: PIPELINE and UNROLL. The UNROLL accelerator method focuses on increasing parallelism by unrolling loops, enabling multiple convolution operations to execute simultaneously. This approach improves data throughput by partitioning arrays to overcome BRAM port limitations, which allows faster data transfers and higher amount of parallel computations. It is also important to mention that this architecture increases the resources usage compared to the PIPELINE accelerator. The latter, by contrast, sets up a sequence of operations that overlap

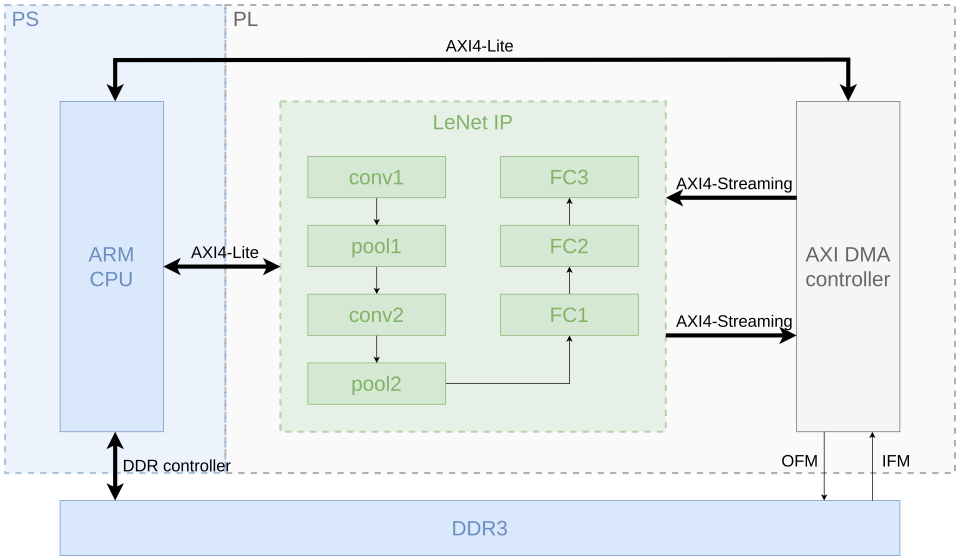


Figure 9: Hardware architecture described by Liang et al. [22]

in execution through pipelining. This allows continuous data flow and greater throughput by minimizing latency between tasks, which is further controlled by adjusting the number of clock cycles needed between iterations for optimal resource use.

3.2 Custom ISA architectures

For the past few years, researchers have also focused on leveraging custom ISAs to further enhance the efficiency of hardware architectures on ML workload. It is spontaneous to understand that a custom ISA is more likely to be more efficient on the specific task it has been customized for than a general purpose ISA. Thus, by carefully adding new instructions, new specialized hardware can be efficiently leveraged. For that matter, RISC-V has been often used in the literature, mainly for its customizability.

Recently, Valente et al. [35] leveraged RISC-V and designed Shaheen, a 9 mm^2, 200mW heterogeneous System-on-Chip (SoC) designed for secure navigation of autonomous nano-UAVs, implemented in 22 nm FDX technology. Its architecture, described in Figure 10, operates across four distinct clock domains, each serving specific functions.

At its core, it features a CVA6 host processor – a 6-stage, in-order 64-bit RISC-V core (RV64GC ISA) with advanced features including virtual memory management, three privilege levels, and hardware virtualization support. The system's memory architecture is built around 1MB of L2 ScratchPad Memory (SPM) and a Tightly-Coupled Data Memory (TCDM) interconnect in the host domain, delivering 64 Gbps read/write performance. This memory serves multiple purposes: data storage, cluster code storage, and facilitating host-cluster communication. A distinctive feature is its HyperRAM memory controller, chosen for its power efficiency, as it is able to interface with over 500MB of external memory. The parallel programmable cluster domain operates as the system's computational powerhouse, featuring 8 Flex-V RV32 cores [26] optimized for DSP and ML workload. These Flex-V cores share 256kB of L1 SPM, providing a bandwidth of 256 Gbps at 500 MHz. The Flex-V cores support various precision formats (FP32, FP16, bfloat16) and use XpulpNN ISA extension [10], an optimized custom ISA for ML, based on XpulpV2 ISA. Among the new instructions included, the Mac&Load (mlsdotp) instruction is a specialized operation that performs two tasks at once: a SIMD dot-product calculation while simultaneously loading new data for the next operation. To achieve this, the design includes six dedicated 32-bit registers (Neural Network Register Files) that store the loaded data during the writeback stage.

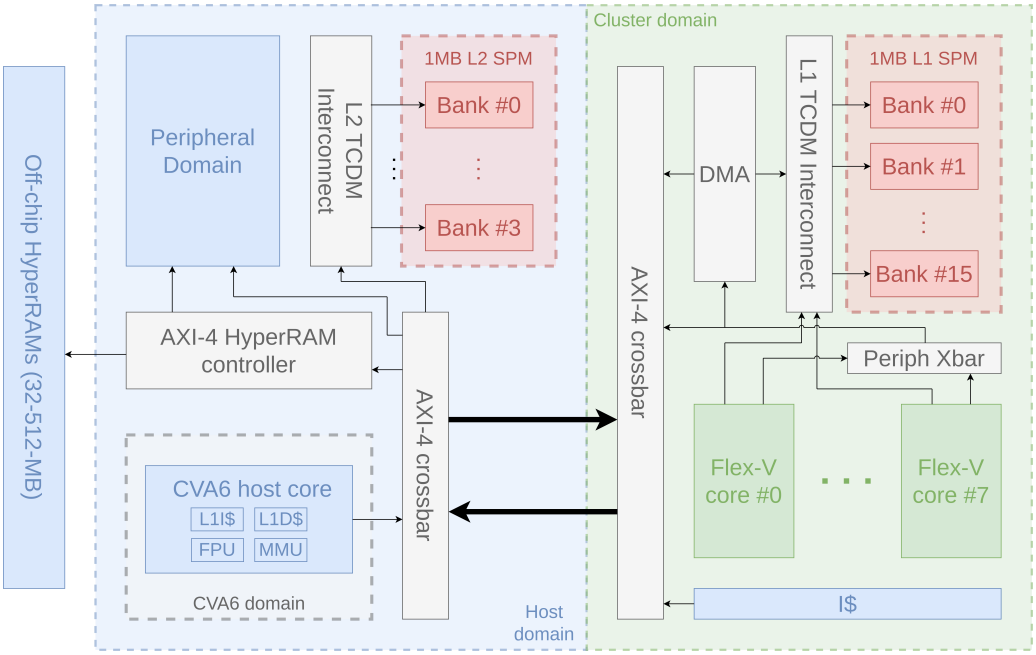


Figure 10: Hardware architecture described by Valente et al. [35]

Furthermore, Askarihemmat et al. [5] designed a DNN accelerator that combines arbitrary precision computation with a RISC-V controller. They also provided a tool that, from a CNN model (in ONNX), generates RISC-V assembly for that specific hardware architecture, especially for the custom ISA that has been developed.

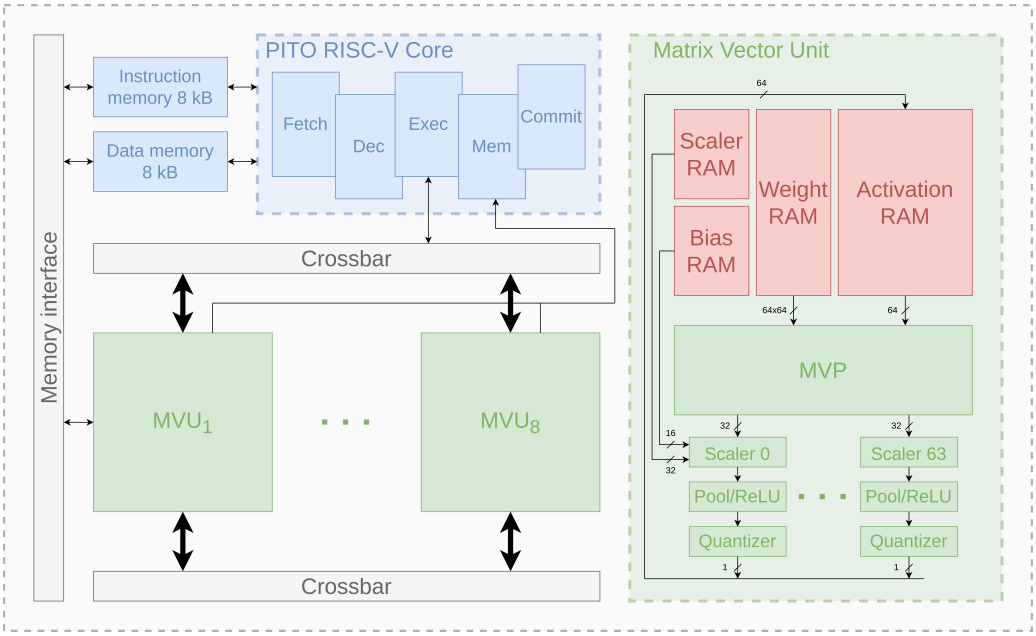


Figure 11: Hardware architecture described by Askarihemmat et al. [5]

Basically, the system is built around Matrix Vector Units (MVUs) that handle the core DNN operations, while the RISC-V CPU, a barrel processor – a processor that switches between threads on every cycle – designed by Askarihemmat et al. [4] is the central controller managing the computational flow. Each MVU functions as a 64-element vector processor that specializes in fundamental DNN operations including GEMV, GEMM, and convolutions. The bit-serial computation strategy is key to enabling arbitrary precision processing – instead of processing entire words at once, the MVUs process data bit by bit, which allows for dynamic

adjustment of numerical precision. The MVUs are structured to optimize throughput through parallel processing capabilities, where each unit can independently handle different data precision configurations. This flexibility means that while one MVU might be processing 8-bit operations, another could simultaneously handle 4-bit or 16-bit computations, depending on the layer's requirements. The memory within each MVU is optimized for efficient data transfers and processing, reducing latency. The architecture employs 8 of these MVUs. The RISC-V controller's role is particularly crucial as it manages the execution of DNN operations by coordinating the MVUs, handling memory transfers, and enabling dynamic precision adjustments without requiring hardware reconfiguration. This integration of RISC-V and custom instructions effectively manages workload distribution and memory organization while maintaining the system's ability to adapt to different DNN architectures.

Then, Yu et al. [40] created a CNN specific ISA extension based on RISC-V. It introduces two key custom instructions: VMAC for vector multiplication operations and VLOAD for vector memory loading operations. These instructions operate on vector registers containing multiple elements, enabling parallel processing of CNN computations. These instructions have been implemented in a zero-riscy core (currently known as Ibex [23], see Figure 12).

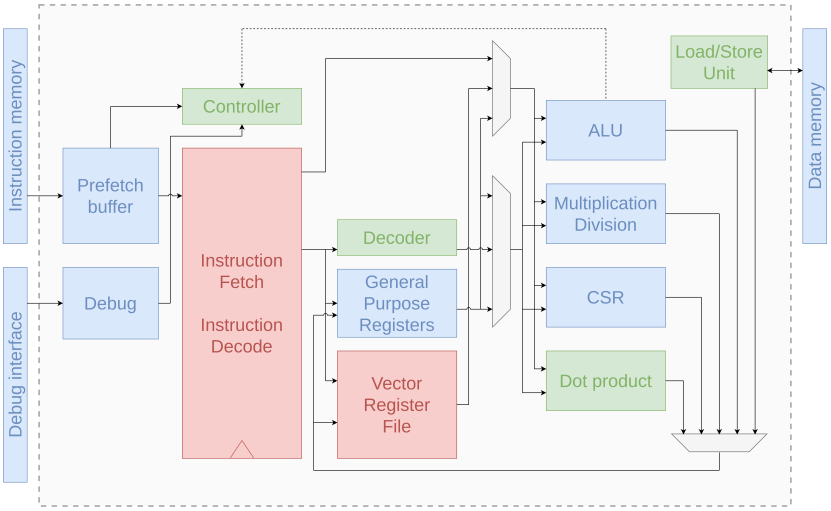


Figure 12: Extended Zero-riscy pipeline introduced by Yu et al. [40]

The zero-riscy core [7] is an open-source architecture that implements the RV32IMC ISA. It is specifically designed and optimized for low-power and energy-efficient applications. In its baseline form, the architecture struggles with CNN workload because these operations require extensive multiplication operations and frequent memory accesses. That is why, including vector registers and a specialized dot-product unit further enhances this architecture on CNN workload, and enables parallel computing capabilities while maintaining resource efficiency.

Kovacevic et al. [19] developed a system, depicted in Figure 13, that employs a dual-processing approach, combining a traditional 5-stage pipelined scalar core, employing RV32IV ISA (meaning it uses the vector extension of RISC-V basic ISA), with a vector core featuring multiple parallel processing vector lanes.

The scalar core manages basic instructions and memory access thanks to a two-level cache system, while the vector core leverages data-level parallelism of ML workload by executing instructions across data arrays and splitting larger vectors into parallel-processed smaller units. The vector core's efficiency is enhanced through several key components. First, the scheduler gets instructions, partially decodes it and hands over the correct signal to the Vector Control Unit (VCU) for coordination, while delivering a stall signal to the scalar core as well. Then the VCU generates control signals that are sent to the drivers. These drivers communicate with every vector lane.

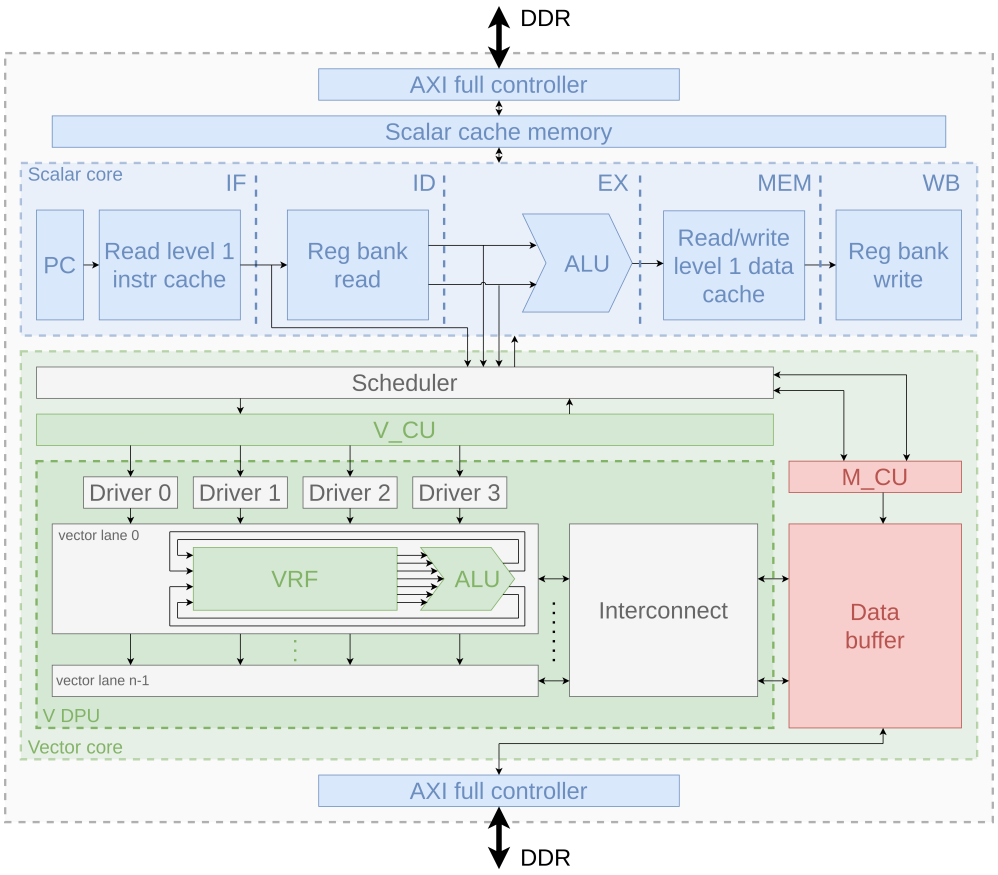


Figure 13: Processor architecture designed by Kovacevic et al. [19]

Synchronization and dependencies between data are handled by the VCU so that Vector Register Files (VRFs) optimally perform computations. These VRFs employ double-pumping and XOR-based techniques for optimal data handling. Finally, a Memory Control Unit (MCU) manages buffers in VRFs and more globally in the vector core, so that vector lanes are ready for next computations as soon as possible.

RISC-V is also leveraged in the design presented by Garofalo et al. [9]. As depicted in Figure 14, they developed the Dustin SoC. This SoC relies on a 16-core RISC-V parallel cluster utilizing a Multiple Instruction Multiple Data (MIMD) configuration. Basically, MIMD is a parallel computing architecture where multiple processors, 16 here, can simultaneously execute different instructions on different data streams.

Then, these cores share a tightly-coupled memory system for efficient data access. The architecture's distinctive feature is its flexible bit-precision support ranging from 2 to 32 bits, implemented through specialized ISA extensions, including customized control registers that enable on-the-fly adaptation of operand formats and enhanced SIMD dot-product execution units that enable mixed-precision computations. Eventually, for power optimization, it incorporates a Vector Lockstep Execution Mode (VLEM) that operates by synchronizing multiple cores to execute identical instruction sequences on different data sets, effectively reducing control overhead and power consumption. It can be selectively activated for data-parallel tasks without compromising the system's ability to handle general-purpose computing in MIMD mode.

By the way, this cluster is stemming from PULP cluster, which is part of a RISC-V ecosystem aiming at, among other things, improving the energy-efficiency of NN on embedded devices. This cluster has also been leveraged by Palossi et al. [27] when they designed a system based on GAP8, a SoC also derived from PULP, meant to be embedded on nano-drones. Before the Dustin SoC, Garofalo et al. [10] worked on RISC-V ISA extensions for accelerating *Quan-*

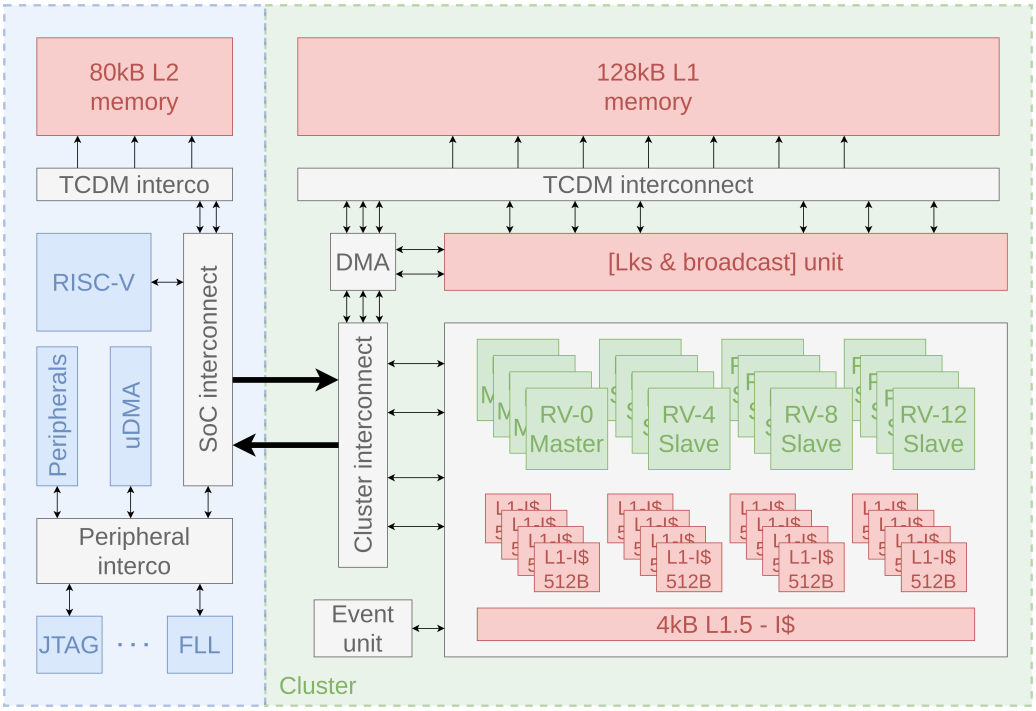


Figure 14: Dustin SoC architecture presented by Garofalo et al. [9]

tized Neural Networks (QNNs). Basically, they created custom SIMD, vector, dot-product and quantization instructions to handle ML workload.

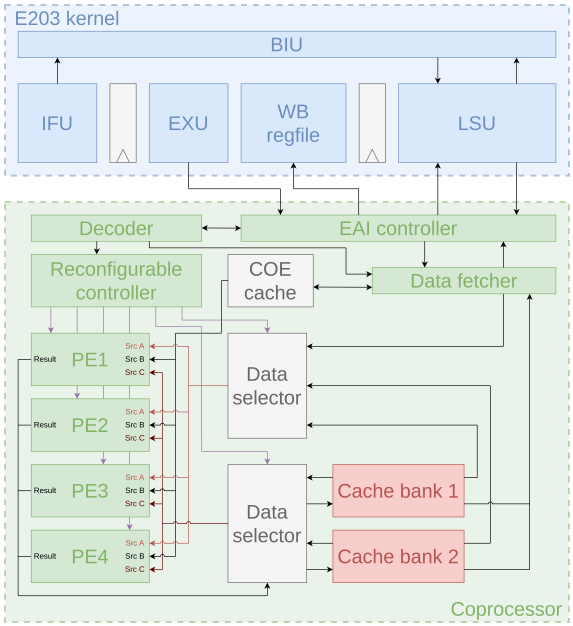
Ultimately, Wu et al. [39] leveraged the E203 core, a 32-bit RISC-V processor core with a 2-level pipeline. This core extends the RV32I ISA with multiplication, division and atomic operations instructions along with 16-bit compression instructions. The CNN accelerator is integrated with the RISC-V CPU as a coprocessor through custom instructions. As depicted in Figure 15a, the design includes a convolution kernel cache (COE Random Access Memory (RAM)), reconfigurable controller, and 4 PEs that can be configured with different parameters to perform convolution, pooling, ReLU, and/or matrix addition, as depicted in Figure 15b.

The crossbar circuit, equipped with input buffers and configuration registers, enables dynamic control of data paths between calculation modules, allowing flexible execution of various ML algorithms. To interact with this hardware, a custom instruction group has been created and added to the ISA. Basically, the coprocessor operates through a sequence of custom instructions. Starting with `acc.rest` for initialization, the system then loads data using `acc.load` instructions for both convolution coefficients and input matrices. The main configuration phase employs multiple `acc.cfg` subgroup of instructions to set up numerous parameters: working modes for PEs, memory management for input and output data, matrix dimensions, convolution parameters, pooling settings, and matrix addition configurations. Finally, the `acc.sc` instruction triggers the actual computation based on all previously configured parameters.

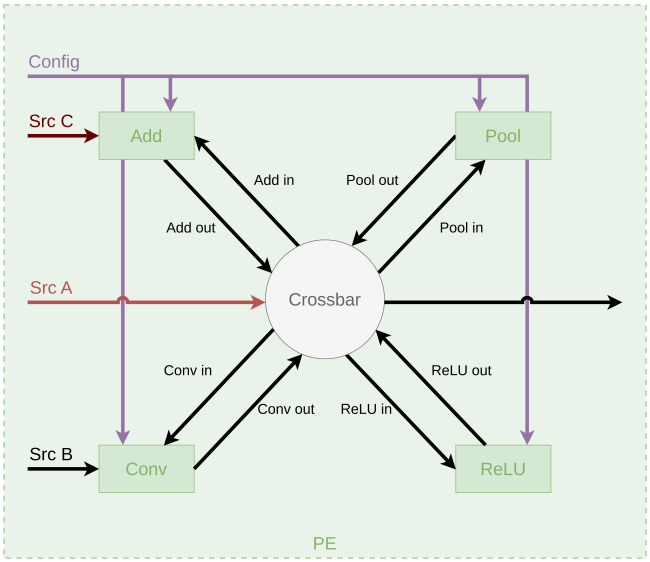
3.3 Data transfers optimization architectures

On top of computations acceleration, data transfers optimizations have also been identified as mandatory when processing ML workload. For that matter, adding buffers in the processing areas or elements has often been the selected solution. But recently, PiM has arisen as a very powerful paradigm. Over the past few years, a couple of architectures have leveraged PiM in various ways.

First, Bavikadi et al. [6] designed a reconfigurable PiM architecture for data intensive applications. A hierarchical overview of this architecture is given in Figure 16. Inside this



(a) Host core E203 and Coprocessor structures.



(b) Processing Element (PE) diagram.

Figure 15: Hardware architecture of host and CNN accelerator developed by Wu et al. [39]

architecture, the acceleration cluster is built around multiple PEs that contain specialized Multi-functional LUT (M-LUT) cores. These cores come in three specialized variants designed for different computational needs as they can be arranged in 3x3, 5x5 or 7x7 arrays. The ALU LUT core processes 4-bit inputs to produce 4-bit outputs through either XOR or AND operations selected via multiplexers, with four cores present in each MAC PE.

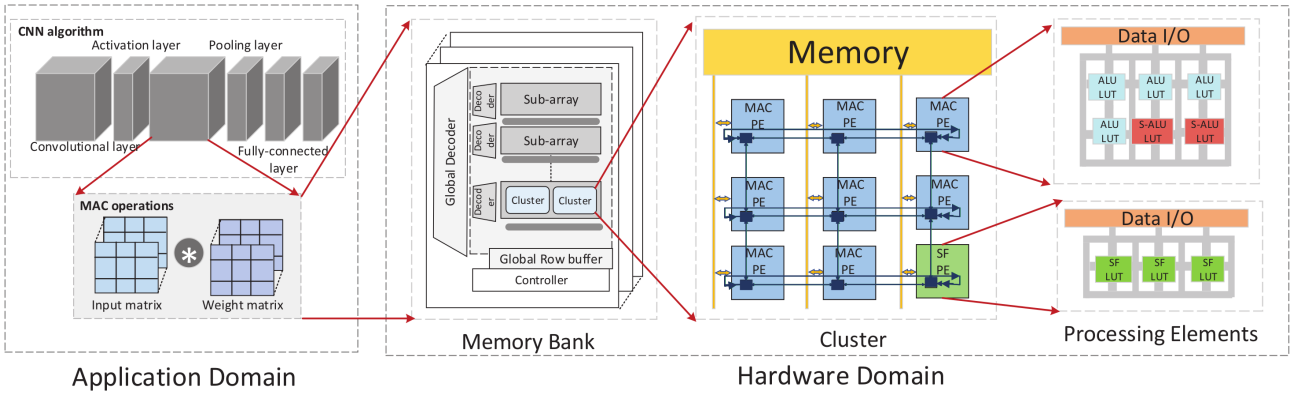


Figure 16: Cluster arrangement and M-LUT core organization inside PEs developed by Bavikadi et al. [6]

The *Special ALU* (S-ALU) LUT core simultaneously performs OR and AND operations, generating an 8-bit output where the upper half contains XOR results and lower half contains AND results, with two cores per MAC PE. Finally, the Special Function (SF) LUT core, exclusively found in SF PEs, handles 8-bit operations for specialized CNN functions including pooling, batch normalization, and various traditional activation functions like sigmoid, hyperbolic, and ReLU. As mentioned previously, these ML-related computations are heavy and specific hardware comes handy to optimize their execution. However, being able to include these PEs directly into the memory is the main feature, as it cuts any data transfer between the off-chip memory and the PEs, therefore reducing the power consumption, latency and other parameters discussed in Section 2. In addition, these clusters are designed to support various

operations across different CNN layers, making the architecture highly adaptable for different ML applications. The use of M-LUT cores enables diverse functional programmable operations on input data, enhancing the flexibility and efficiency of the overall system.

Then, Zhang et al. [42] developed PIMCA, a programmable PiM accelerator for low-precision (1-2 bits) DNN inference, leveraging a unique 10T1C (10 Transistors, 1 Capacitor) bitcell architecture that performs MAC operations through capacitive coupling in an analog mixed-signal domain. This bitcell enhances a standard 6T Static Random Access Memory (SRAM) cell by adding two transmission gates and a capacitor placed above the transistors. This design eliminates threshold voltage drops and improves robustness against process variations compared to previous designs. The accelerator contains 108 In-Memory Computing (IMC) SRAM macros, each featuring 256 × 128 bitcells that can activate all rows simultaneously, enabling GEMV to be completed in a single cycle. The hardware includes a custom six-stage pipeline, described Figure 17, and a specialized ISA that supports hardware loop functionality, reducing program size by up to 73%. By integrating arithmetic functions directly within the memory, PIMCA eliminates the traditional bottlenecks of memory accesses and enables simultaneous data access and computation. To handle computational errors from analog operations, the system incorporates an error-aware training framework that models and accounts for these variations during the training process.

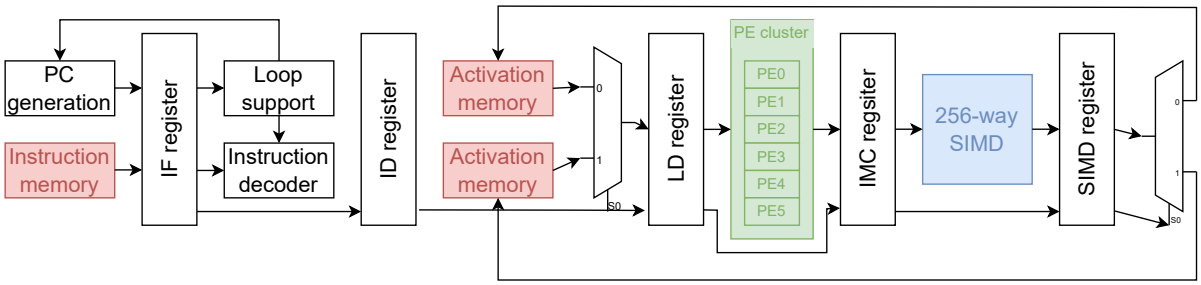


Figure 17: Custom six-stage pipeline of the PIMCA accelerator developed by Zhang et al. [42]

Furthermore, Garofalo et al. [8] leveraged a parallel cluster that has already been discussed in Section 3.2: the PULP platform. Nonetheless, this time, they included a PiM accelerator. As shown in Figure 18, the architecture combines 8 RISC-V cores with an In-Memory computing Accelerator (IMA) and digital accelerators in a tightly-coupled cluster configuration (included in the *accelerators domain* module). The IMA component leverages Analog In-Memory Computing (AIMC) principles by performing computations directly within memory arrays.

This is particularly optimized for GEMV. The system operates by mapping operands onto memory array crossbars, which allows for computational operations to occur within the memory itself rather than shuttling data between separate processing and memory units, which obviously increases latency and decreases energy-efficiency. This approach utilizes either charge-based or resistance-based memory technologies.

Although PiM is a powerful feature to overcome data transfers bottlenecks, some work has attempted to tackle this issue by using smart memory management strategies. For instance, Wei et al. [38] developed a Layer Conscious Memory Management (LCMM) framework that employs a specialized memory allocation algorithm called DNN Knapsack (DNNK) which utilizes a dynamic programming approach to allocate physical buffers, along with buffer sharing strategies. The framework incorporates buffer prefetching and splitting techniques to reduce latency, while performing liveness analysis for feature tensors and prefetching for weight tensors to minimize unnecessary off-chip transfers.

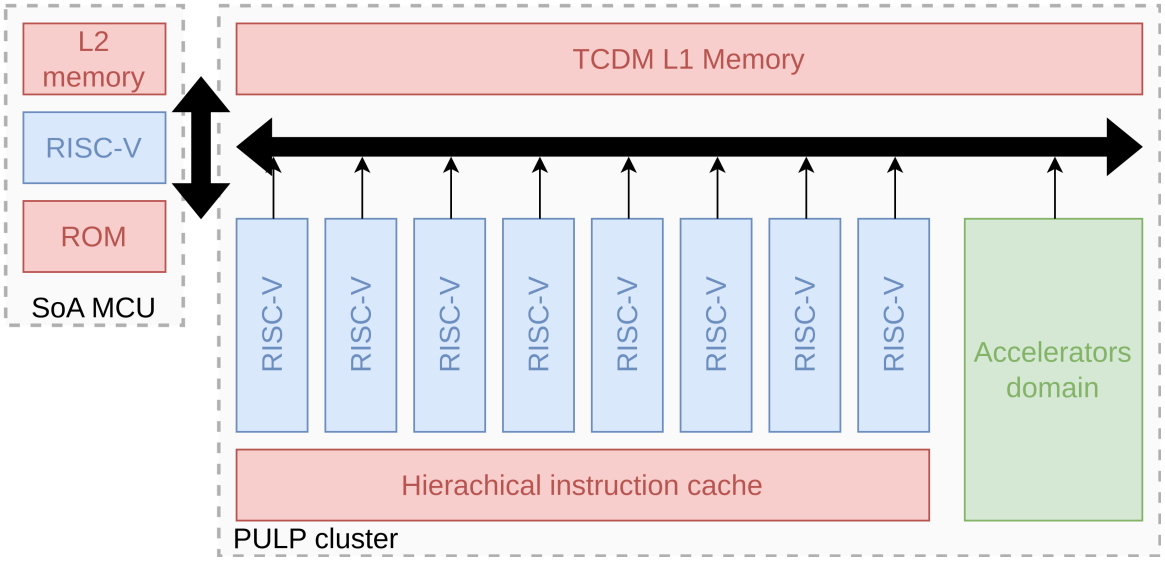
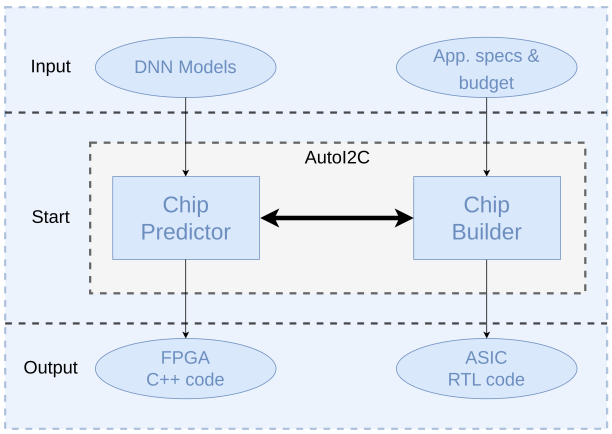


Figure 18: Overview of the PULP cluster, integrating the PiM accelerator created by Garofalo et al. [8]

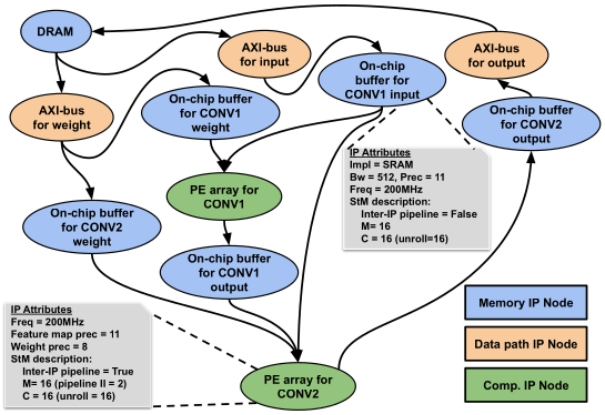
3.4 Co-design approaches

As hardware performance is tailored by ML model it is meant to process, researchers have recently focused on developing frameworks that generate hardware depending on the ML model or algorithm it is inferring. These approaches often leverages co-design, referring to the simultaneous optimization of hardware (in our case, the dedicated hardware architecture) and software (in our case, the ML model).

First, Zhang et al. [43] developed AutoAI2C, a framework that achieves co-design through its Chip Predictor and Chip Builder components.



(a) AutoAI2C overview and specifications.



(b) Graph-based design space description on a heterogeneous accelerator architecture to accelerate ResNet.

Figure 19: AutoAI2C, a framework developed by Zhang et al. [43]

They work together, as depicted in Figure 19a, to explore and evaluate different hardware configurations while considering the specific requirements of ML models from frameworks like PyTorch. The system employs a graph-based representation that serves as a comprehensive "one-for-all" design space representation. It operates across three key abstraction levels: architecture, hardware components, and data mappings. The graph representation illustrating the design space description to accelerate the residual block in ResNet [12] for a specific hardware

is given Figure 19b. Thanks to this representation, the Chip Builder performs effective design space exploration, and uses the graph-based information to identify and optimize promising hardware configurations, ultimately leading to improved accelerator designs.

Design space exploration is also leveraged by Li et al. [21], but they exploit a Reinforcement Learning (RL)-based technique to fine-tune the hardware architecture. Basically, their strategy builds efficient hardware for Recurrent Neural Networks (RNNs) inference through a two-stage optimization approach. In the first stage, it uses a genetic algorithm to narrow down the design space by specifically optimizing hardware on area constraints. This first step helps eliminating impractical design choices and focuses on relevant hardware configurations. Following this, the second stage applies RL to optimize latency and power consumption within the reduced subspace resulting from the previous stage. This method continuously evaluates designs against FPGA resource constraints, adjusting allocations as needed, and it provides the best hardware solution without violating hardware constraints.

While the two strategies previously described only encompass specific PEs and logic in their design space, some strategies make use of the PiM paradigm as well. For instance, Han et al. [11] have set up the CoMN platform described in Figure 20. It integrates four main components that work in concert to optimize Non Volatile Memory (NVM)-based CNN accelerators. At its core, the platform features a mapper that automatically handles the translation of CNN models to PiM chips, carefully optimizing various parameters including pipeline configurations and weight transformations to ensure optimal performance.

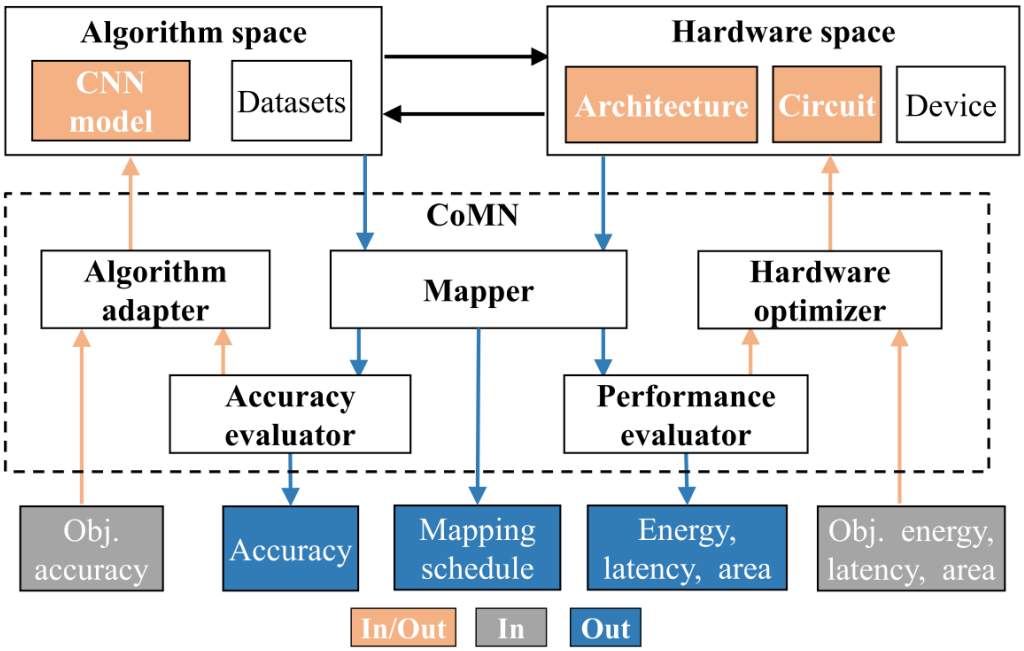


Figure 20: Overview of the CoMN platform, described by Han et al. [11]

This works closely with comprehensive evaluation systems (performance evaluator and accuracy evaluator) that simultaneously monitor and assess multiple critical metrics, including CNN accuracy, energy efficiency, processing latency, and physical area requirements. To address potential hardware-induced accuracy degradation, the platform incorporates an algorithm adapter that enables CNN weight retraining, effectively fine-tuning the network to maintain high accuracy despite hardware constraints. The fourth key component is a hardware optimizer that facilitates early-stage circuit design exploration, allowing designers to make crucial architectural decisions before committing to final implementations. Together, these functionalities create a robust co-design environment that effectively manages the intricate interdependencies across various design levels.

Ultimately, Verma et al. [36] mixed up both strategies and incorporate custom PEs and PiM. Basically, they created a hardware-software co-design approach, called *EXTREM-EDGE* and explained in Figure 22, to add custom extensions to RISC-V ISA for designing scalable and flexible ML processor architecture. The system implements custom ISA extensions through two approaches: "Top-down" extensions are derived from analyzing common operations that frequently occur in AI workload, essentially focusing on the algorithmic needs of AI applications, and "Bottom-up" extensions are designed specifically to leverage the capabilities of the hardware AI Functional Units (AFUs), ensuring that the hardware-specialized features are directly accessible through the instruction set.

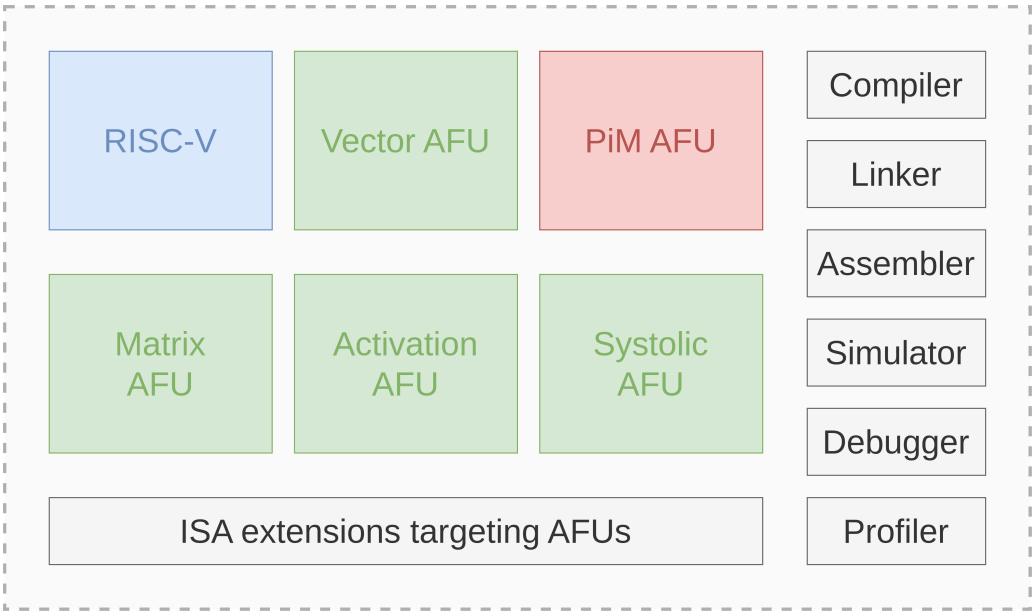


Figure 21: Top-level overview of EXTREM-EDGE presented by Verma et al. [36]

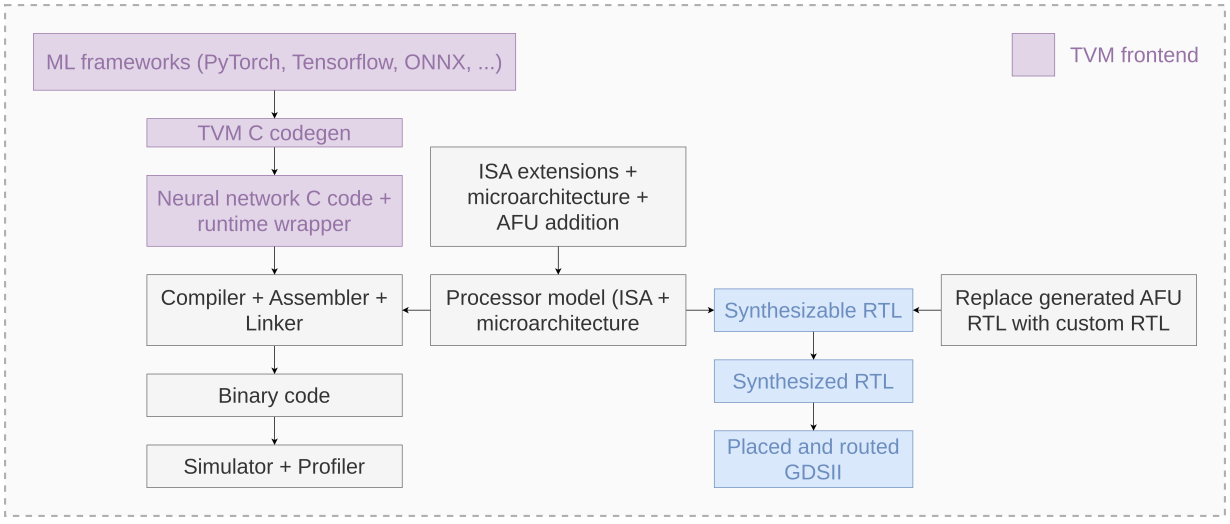


Figure 22: Detailed overview of EXTREM-EDGE backend, designed by Verma et al. [36]

As depicted in Figure 21, AFUs are available in various configurations. For instance, MAC AFUs facilitate matrix-related computations by performing MAC operations, PiM AFUs act as traditional register files, with the added capability of executing computations directly within the register file. These AFUs are relevant because they do not introduce overheads as computational AFUs or PiM AFUs act as standard ALUs or register files. Therefore, additional communication buses are not required.

4 Discussion

As described in Section 3, various strategies have been leveraged to design architectures optimized for the inference of ML algorithms on embedded devices. To grasp the role of the developed technical features, we classify them as follows:

- Special PEs: encompass dedicated components optimized for standard ML functions to efficiently handle intensive computational workload,
- PiM components: include special memory components that perform computations within the memory itself,
- Custom ISAs: gather all the instructions developed for ML hardware acceleration,
- Hardware parallelism: hardware architectural decisions that enhance the parallelisms of accelerators, mainly including hardware duplication,
- On-chip memory: refers to each efficient memory management technique implemented.

Thanks to these categories, the developed architectures can be evaluated to assess whether they address the challenges discussed in Section 2. Then, all the features described in each architecture presented in Section 3 are gathered in Tables 1 and 2. According to these tables, it appears that 17 out of 19 architectures implemented their own specific PEs to optimize ML-related computations. Thus, that feature almost appears as unanimous and seems to be a powerful strategy to increase the throughput and reduce latency. However, these PEs are almost always different and depend on the types of models the architecture is meant to accelerate, which is also true for the hardware parallelism strategies. Then, we understand that PiM has captured the researchers attention for the past couple of years and that more and more PiM-based architectures arise to achieve ML hardware acceleration. Furthermore, it is clear that a lot of effort has been put into leveraging RISC-V ISA and its ability to easily be extended, as 7 out of 8 architectures that use custom ISAs are RISC-V based. Then, almost every architecture (18 out of 19) added on-chip memory buffers. Therefore, this feature appears unavoidable to significantly cut data movements bottlenecks and on-chip storage limitations. Eventually, the "multi-model" column teaches us that almost every architecture is not meant to be efficient on a specific model but, either on a specific type of model with a range of acceptable model sizes, or on various types of ML models. For instance, some architectures are meant to be efficient on several types of models: NNs (*e.g.* feed-forward networks, CNNs and/or RNNs), but also on RFs or SVMs. However, some architectures may be very efficient on a range of CNNs models. For example, this range could be a list of data types for the model's parameters, as well as a range of convolution kernel sizes. Eventually, while the "data precision" column confirms the versatility of these hardware architectures, this column also states that more than 32 bits of precision for the models parameters and weights is not mandatory, that fixed-point and floating-point arithmetic can be implemented, and that software or hardware quantization is almost always employed, appearing as an unavoidable feature in embedded hardware acceleration.

Furthermore, there is no feature addressing all the challenges discussed in Section 2. Basically, it is of paramount importance to understand that each feature in each category will play a different role in addressing these challenges, and only a combination of features from a lot of different categories allows the hardware to be efficient in all the ways that have been discussed previously. As a matter of fact, some features that may enhance the performance (*e.g.* throughput) of an architecture might hinder the power consumption or the adaptability of the architecture. Consequently, a fully optimized hardware architecture capable of achieving

Architectures	Special Processing Elements	Processing-in-Memory	Custom ISAs	Hardware parallelism	On-chip memory	Multi-model	Data precision
Taka et al. [34] - Versal ACAP	AI Engines	✗	Vector instructions (SIMD), load/store	Array of AIE	On-chip buffers in PL	✓	8-bit INT (software quantization)
Taka et al. [34] - Stratix 10 NX	Tensor Blocks (with custom operating modes)	✗	✗	✗	Tensor buffers	✓	32-bit INT, 32-bit Floating-Point
Liang et al. [22]	Convolution, activation, pooling and fully connected operations	✗	✗	✗	All parameters on chip	✗	32-bit Floating-Point
Jain et al. [15]	MAC units, non-linear functions generator, pooling units	✗	"ucode" instructions	✗	512 kB L2 SRAM, 512 kB eMRAM	✓	8/4/2-bit INT (hardware quantization)
Song et al. [31]	CNN Kernels: conv2d, concat, maxpool, optimized DSP	✗	✗	3 CNN kernels	I/O buffers for each CNN Kernel (URAM), Weight buffers (BRAM)	✓	8-bits Fixed-Point (hybrid quantization)
Valente et al. [35]	Flex-V cores [26]: 16/8/4/2-bit dot-products blocks	✗	MAC&Load fused	8 Flex-V cores	16x16kB SRAM banks shared by RV32 cores	✓	Mixed-precision (software quantization)
Askarihemmat et al. [5]	GEMV hardware calculation, pooling, ReLU, scaler, quantizer	✗	✗	8 MVUs	Weights, activation, scalers, biases buffers	✓	16- to 1-bit Fixed-Point (hardware quantization)
Yu et al. [40]	Dot-product unit	✗	Data operation and data transfers custom instructions	✗	Vector Register File	✓	16-bit INT
Kovacevic et al. [19]	✗	✗	✗	Parametrizable number of vector lanes	Vector Register File per Vector lane and Memory Control Unit	✓	32-bit Fixed-Point
Garofalo et al. [9]	16/8/4/2-bit dot-products blocks	✗	RV32I ISA extended for 16-bit and 8-bit dot-products	16 cores	32 banks of TCDM (128 kB)	✓	2- to 32-bit Fixed-Point

Table 1: Summary of ML hardware acceleration architectures (part 1/2).

Architectures	Special Processing Elements	Processing-in-Memory	Custom ISAs	Hardware parallelism	On-chip memory	Multi-model	Data precision
Bavikadi et al. [6]	Programmable Architectures Functional Units (PFUs): M-LUTs containing MAC units, Special Functions Units	PFUs in the memory bank	✗	Multiple clusters, multiple cores	DRAM sub-arrays	✓	8/4-bit Fixed-Point
Zhang et al. [42]	PEs (18 IMC SRAM with adder trees for convolution & fully-connected layers), 256-way SIMD (for non-MAC operations)	IMC Macro: capacitive-coupling memory cell	Custom ISA: 10 72-bit instructions	6 PEs, 256-way SIMD block	Multiple registers, Activation memory (2x6 banks, each bank stores 1024x128 bits)	✗	2/1-bit INT (software quantization)
Zhang et al. [43]	PEs: adder trees, MAC IPs, Convolutions	✗	✗	Arrays of PEs, NoC	On-chip buffers for each PE array, buffers inside PEs	✓	Mixed-precision (fully parametrizable)
Han et al. [11]	✗	non-volatile CiM (nvCiM) Macros	✗	Array of Macros	nvCiM array with parametrizable size	✓	8-bit Fixed-Point (Quantization using ADCs)
Verma et al. [36]	AFU: GEMV, GEMM, Activation, etc.	PiM AFU	Custom extension of RISC-V ISA	Array of AFUs	✗	✓	32-bit Fixed-Point (Software quantization using TVM)
Wu et al. [39]	PE: add, pooling, convolution, ReLU (crossbar selection)	✗	RISC-V ISA + 15 coprocessor instructions	4 PEs	RAM buffers banks, convolution kernel cache RAM	✓	32-bit INT
Palossi et al. [27]	Enhanced DSP (e.g. fixed-point dot-product)	✗	RV32IMC + MAC, SIMD DSP and bit manipulation instructions	8 optimized RISC-V cores	64 kB shared L1 memory, 512kB L2 memory	✓	32-bit Fixed-Point
Medus et al. [24]	NPEs (VMM operations), 1 activation function block (ReLU, Sigmoid, tanh)	✗	✗	Amount of NPEs parametrizable	BRAM36 memory blocks for weights	✓	Up to 18-bit word length
Khabbazan et al. [17]	MAC blocks, ReLU blocks	✗	✗	Loops unrolled over hardware	36kB BRAM (Kernel memory), I/O buffers	✓	(hardware quantization)

Table 2: Summary of ML hardware acceleration architectures (part 2/2).

equal efficiency across all types of ML algorithms or applications is an unrealistic ideal. Indeed, trade-offs have to be made to reach an optimal design, which itself varies depending on the specific applications envisioned for the hardware.

For instance, considering an UAV as the edge device, applications may require the use of versatile ML algorithms (*e.g.* ensemble methods may leverage CNNs, RFs and/or SVMs for object detection, RNNs for target tracking). Hence, depending on each model type, size and parameters, hardware features may be mandatory to enhance throughput and adaptability. In that particular case, hardware parallelism features such as multiple cores, multiple PEs and an optimized communication method between the PE is relevant.

In addition, a large amount of on-chip memory and high-throughput data buses may be required so that switching from a model type to another does not increase the latency of the architecture. However, some other features must not be included when the embedded ML models are that much versatile. For example, including very specific hardware blocks (*e.g.* 2x2 convolution kernel acceleration blocks) may be useful on specific models (*e.g.* CNNs with 2x2 convolution kernels), but useless or inefficient for, first, other models of the same type (*e.g.* CNNs with 3x3 convolution kernels) and, second, other types of models as well. Consequently, an unwised choice in the set of features may decrease the area efficiency of the hardware architecture, as well as the throughput, hindering the overall performance of the system.

Architecture	**Device/Technology**	**Data type**	**Frequency**	**Power**	**Throughput**	**Energy-efficiency**
Taka et al. [34] (Versal ACAP)	Versal VC1902	INT8	290 MHz	82 W	77 TOPS	0.94 TOPS/W
Taka et al. [34] (Stratix 10 NX)	Intel Stratix 10 NX	INT8	327 MHz	52.5 W	66.94 TOPS	1.275 TOPS/W
Liang et al. [22]	ZYNQ 7000	Fixed-Point 32 bits	100 MHz	2.193 W 2.029 W	935 Frame/s 62.4 Frame/s	426 Frame/s/W (PIPELINE) 30.8 Frame/s/W (UNROLL)
Jain et al. [15]	22FDX	INT8 INT4 INT2	5 MHz	237 μW 197 μW 197 μW	0.586 GOPS 1.17 GOPS 2.35 GOPS	2.47 TOPS/W 5.94 TOPS/W 11.9 TOPS/W
Song et al. [31]	Virtex UltraScale+ VU9P	FP8	200 MHz	29.68 W	2.7648 TOPS	0.0929 TOPS/W
Valente et al. [35]	22 nm FD-SOI	INT8 INT4 INT2 FP32 FP16	500-600 MHz	195 mW	26 GOPS 50 GOPS 90 GOPS 4 GFLOPS 7.9 GFLOPS	135 GOPS/W 256 GOPS/W 462 GOPS/W 20.5 GFLOPS/W 40.5 GFLOPS/W
Askarihemmat et al. [5]	Alveo U250	Fixed-Point 16 bits	250 MHz	46.5 mW	2296 Frame/s	106.8 Frame/s/W
Yu et al. [40]	-	INT8	-	-	2.48x−2.82x speed-up ratio	-
Kovacevic et al. [19]	Zedboard	Fixed-Point 32 bits	100 MHz	1.301 W	1.13 conv/s	0.87 conv/s/W
Garofalo et al. [9]	CMOS 65 nm	INT8 INT4 INT2	205 MHz	156 mW	15 GOPS 30 GOPS 58 GOPS	303 GOPS/W 570 GOPS/W 1152 GOPS/W
Bavikadi et al. [6]	TSMC 28 nm	Fixed-Point 8 bits	-	0.87 mW	45.9 Frame/s	52.6 kFrame/s/W
Zhang et al. [42]	28 nm	INT1	42 MHz	124 mW	49 TOPS	437 TOPS/W
Zhang et al. [43]	ZYNQ 7000 (ZC706)	INT16	200 MHz	5.89 W	27.4 Frame/s	4.65 Frame/s/W
Han et al. [11]	22 nm	INT8	1.2 GHz	-	-	8.33 kFrame/s/W
Verma et al. [36]	-	INT64	-	-	20 Mcycles/Frame	-
Wu et al. [39]	Xilinx XC7A100T	INT32	-	-	6.27x speed-up ratio	-
Palossi et al. [27]	GAP8 SoC	Fixed-Point 32 bits	100 MHz	566 mW	7.26 Frame/s	12.8 Frame/s/W
Medus et al. [24]	Virtex 7	Fixed-Point 12 bits	550 MHz	-	1.98 GOPS	-
Khabbazan et al. [17]	ZYNQ 7000	Fixed-Point 8 bits	160 MHz	1.77 W	40.96 GOPS	23.14 GOPS/W

Table 3: Hardware acceleration evaluation of the architectures described in Section 3

This is where co-design approaches, allowing to build hardware that depends on the software and on the user's needs for their specific application can, sometimes, arise as powerful strategies to find the trade-offs that meet our requirements.

5 Conclusion

For the past few years, ML has increasingly shifted from traditional server-based processing to embedded solutions to address challenges such as latency or unreliable communication links, for UAV-based IDSs for instance. In this work, we aim at examining the challenges of embedded ML hardware acceleration, focusing on versatile, scalable architectures capable of supporting multi-model applications efficiently. Addressing these challenges requires balancing multiple factors: performance, energy usage, flexibility for different applications, and computational accuracy. The field continues its evolution through various approaches, from specialized PEs to PiM architectures. Supporting multiple applications can also be challenging and is addressed through reconfigurability and adaptability strategies. For that matter, recent developments in co-design frameworks and automated tools are promising. Optimizing both hardware and ML algorithms together seems to pave the way for more efficient and adaptable solutions for future embedded ML applications.

References

[1] Ahmed F. AbouElhamayed, Angela Cui, Javier Fernandez-Marques, Nicholas D. Lane, and Mohamed S. Abdelfattah. PQA: Exploring the Potential of Product Quantization in DNN Hardware Acceleration. *ACM Transactions on Reconfigurable Technology and Systems*, page 3656643, April 2024.

[2] AMD/Xilinx. Ai engine api user guide, 2022. `https://www.xilinx.com/htmldocs/xilinx2022_1/aiengine_api/aie_api/doc/index.html`.

[3] AMD/Xilinx. Vitis high-level synthesis user guide (ug1399)„ 2022.

[4] MohammadHossein AskariHemmat, Olexa Bilaniuk, Sean Wagner, Yvon Savaria, and Jean-Pierre David. RISC-V Barrel Processor for Deep Neural Network Acceleration. In *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5, Daegu, Korea, May 2021. IEEE.

[5] Mohammadhossein Askarihemmat, Sean Wagner, Olexa Bilaniuk, Yassine Hariri, Yvon Savaria, and Jean-Pierre David. BARVINN: Arbitrary Precision DNN Accelerator Controlled by a RISC-V CPU. In *Proceedings of the 28th Asia and South Pacific Design Automation Conference*, pages 483–489, January 2023.

[6] Sathwika Bavikadi, Purab Ranjan Sutradhar, Amlan Ganguly, and Sai Manoj Pudukotai Dinakarrao. Reconfigurable Processing-in-Memory Architecture for Data Intensive Applications. In *2024 37th International Conference on VLSI Design and 2024 23rd International Conference on Embedded Systems (VLSID)*, pages 222–227, Kolkata, India, January 2024. IEEE.

[7] Pasquale Davide Schiavone, Francesco Conti, Davide Rossi, Michael Gautschi, Antonio Pullini, Eric Flamand, and Luca Benini. Slow and steady wins the race? A comparison of ultra-low-power RISC-V cores for Internet-of-Things applications. In *2017 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, pages 1–8, Thessaloniki, September 2017. IEEE.

[8] Angelo Garofalo, Gianmarco Ottavi, Francesco Conti, Geethan Karunaratne, Irem Boybat, Luca Benini, and Davide Rossi. A Heterogeneous In-Memory Computing Cluster For Flexible End-to-End Inference of Real-World Deep Neural Networks, January 2022. arXiv:2201.01089 [cs].

[9] Angelo Garofalo, Gianmarco Ottavi, Alfio Di Mauro, Francesco Conti, Giuseppe Tagliavini, Luca Benini, and Davide Rossi. A 1.15 TOPS/W, 16-Cores Parallel Ultra-Low Power Cluster with 2b-to-32b Fully Flexible Bit-Precision and Vector Lockstep Execution Mode. In *ESSCIRC 2021 - IEEE 47th European Solid State Circuits Conference (ESSCIRC)*, pages 267–270, Grenoble, France, September 2021. IEEE.

[10] Angelo Garofalo, Giuseppe Tagliavini, Francesco Conti, Davide Rossi, and Luca Benini. XpulpNN: Accelerating Quantized Neural Networks on RISC-V Processors Through ISA Extensions. In *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 186–191, Grenoble, France, March 2020. IEEE.

[11] Lixia Han, Renjie Pan, Zheng Zhou, Hairuo Lu, Yiyang Chen, Haozhang Yang, Peng Huang, Guangyu Sun, Xiaoyan Liu, and Jinfeng Kang. CoMN: Algorithm-Hardware Co-Design Platform for Nonvolatile Memory-Based Convolutional Neural Network Accelerators. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 43(7):2043–2056, July 2024.

[12] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition, December 2015.

[13] Yunxiang Hu, Yuhao Liu, and Zhuovuan Liu. A Survey on Convolutional Neural Network Accelerators: GPU, FPGA and ASIC. In *2022 14th International Conference on Computer Research and Development (ICCRD)*, pages 100–107, Shenzhen, China, January 2022. IEEE.

[14] RISC-V International. *RISC-V Specifications*. RISC-V International, 2024.

[15] Vikram Jain, Sebastian Giraldo, Jaro De Roose, Linyan Mei, Bert Boons, and Marian Verhelst. TinyVers: A Tiny Versatile System-on-Chip With State-Retentive eMRAM for ML Inference at the Extreme Edge. *IEEE Journal of Solid-State Circuits*, 58(8):2360–2371, August 2023.

[16] Rupinder Kaur, Arghavan Asad, and Farah Mohammadi. A Comprehensive Review of Processing-in-Memory Architectures for Deep Neural Networks. *Computers*, 13(7):174, July 2024.

[17] Bahareh Khabbazan and Sattar Mirzakuchaki. Design and Implementation of a Low-Power, Embedded CNN Accelerator on a Low-end FPGA. In *2019 22nd Euromicro Conference on Digital System Design (DSD)*, pages 647–650, Kallithea, Greece, August 2019. IEEE.

[18] Kasem Khalil, Bappaditya Dey, Ashok Kumar, and Magdy Bayoumi. Adaptive Hardware Architecture for Neural-Network-on-Chip. In *2022 IEEE 65th International Midwest Symposium on Circuits and Systems (MWSCAS)*, pages 1–4, Fukuoka, Japan, August 2022. IEEE.

[19] Nikola Kovacevic, Dorde Miseljic, and Aleksa Stojkovic. RISC-V vector processor for acceleration of machine learning algorithms. In *2022 30th Telecommunications Forum (TELFOR)*, pages 1–4, Belgrade, Serbia, November 2022. IEEE.

[20] Andrey Kuzmin, Markus Nagel, Mart van Baalen, Arash Behboodi, and Tijmen Blankevoort. Pruning vs Quantization: Which is Better? *Advances in Neural Information Processing Systems 36 (NeurIPS 2023)*, 2023.

[21] Qingpeng Li, Jian Xiao, and Jizeng Wei. GRMD: A Two-Stage Design Space Exploration Strategy for Customized RNN Accelerators. *Symmetry*, 16(11):1546, November 2024.

[22] Yong Liang, Junwen Tan, Zhisong Xie, Zetao Chen, Daoqian Lin, and Zhenhao Yang. Research on Convolutional Neural Network Inference Acceleration and Performance Optimization for Edge Intelligence. *Sensors*, 24(1):240, December 2023.

[23] lowRISC. Ibex risc-v core, 2024. `https://github.com/lowRISC/ibex`.

[24] Leandro D. Medus, Taras Iakymchuk, Jose Vicente Frances-Villora, Manuel Bataller-Mompean, and Alfredo Rosado-Munoz. A Novel Systolic Parallel Hardware Architecture for the FPGA Acceleration of Feedforward Neural Networks. *IEEE Access*, 7:76084–76103, 2019.

[25] Tamador Mohaidat and Kasem Khalil. A Survey on Neural Network Hardware Accelerators. *IEEE Transactions on Artificial Intelligence*, 5(8):3801–3822, August 2024.

[26] Alessandro Nadalini, Georg Rutishauser, Alessio Burrello, Nazareno Bruschi, Angelo Garofalo, Luca Benini, Francesco Conti, and Davide Rossi. A 3 TOPS/W RISC-V Parallel Cluster for Inference of Fine-Grain Mixed-Precision Quantized Neural Networks, July 2023.

[27] Daniele Palossi, Antonio Loquercio, Francesco Conti, Eric Flamand, Davide Scaramuzza, and Luca Benini. A 64-mW DNN-Based Visual Navigation Engine for Autonomous Nano-Drones. *IEEE Internet of Things Journal*, 6(5):8357–8371, October 2019.

[28] Albert Reuther, Peter Michaleas, Michael Jones, Vijay Gadepally, Siddharth Samsi, and Jeremy Kepner. AI and ML Accelerator Survey and Trends. In *2022 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–10, Waltham, MA, USA, September 2022. IEEE.

[29] Alejandra Sanchez-Flores, Lluc Alvarez, and Bartomeu Alorda-Ladaria. A review of CNN accelerators for embedded systems based on RISC-V. In *2022 IEEE International Conference on Omni-layer Intelligent Systems (COINS)*, pages 1–6, Barcelona, Spain, August 2022. IEEE.

[30] Cristina Silvano, Daniele Ielmini, Fabrizio Ferrandi, Leandro Fiorin, Serena Curzel, Luca Benini, Francesco Conti, Angelo Garofalo, Cristian Zambelli, Enrico Calore, Sebastiano Fabio Schifano, Maurizio Palesi, Giuseppe Ascia, Davide Patti, Nicola Petra, Davide De Caro, Luciano Lavagno, Teodoro Urso, Valeria Cardellini, Gian Carlo Cardarilli, Robert Birke, and Stefania Perri. A Survey on Deep Learning Hardware Accelerators for Heterogeneous HPC Platforms, July 2024.

[31] Qingzeng Song, Jiabing Zhang, Liankun Sun, and Guanghao Jin. Design and Implementation of Convolutional Neural Networks Accelerator Based on Multidie. *IEEE Access*, 10:91497–91508, 2022.

[32] Febin Sunny, Amin Shafiee, Abhishek Balasubramaniam, Mahdi Nikdast, and Sudeep Pasricha. OPIMA: Optical Processing-in-Memory for Convolutional Neural Network Acceleration. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 43(11):3888–3899, November 2024.

[33] Endri Taka, Aman Arora, Kai-Chiang Wu, and Diana Marculescu. MaxEVA: Maximizing the Efficiency of Matrix Multiplication on Versal AI Engine, November 2023. arXiv:2311.04980 [cs].

[34] Endri Taka, Dimitrios Gourounas, Andreas Gerstlauer, Diana Marculescu, and Aman Arora. Efficient Approaches for GEMM Acceleration on Leading AI-Optimized FPGAs, April 2024.

[35] Luca Valente, Alessandro Nadalini, Asif Veeran, Mattia Sinigaglia, Bruno Sa, Nils Wistoff, Yvan Tortorella, Simone Benatti, Rafail Psiakis, Ari Kulmala, Baker Mohammad, Sandro Pinto, Daniele Palossi, Luca Benini, and Davide Rossi. A Heterogeneous RISC-V based SoC for Secure Nano-UAV Navigation, January 2024.

[36] Vaibhav Verma, Tommy Tracy Ii, and Mircea R. Stan. EXTREM-EDGE—EXtensions To RISC-V for Energy-efficient ML inference at the EDGE of IoT. *Sustainable Computing: Informatics and Systems*, 35:100742, September 2022.

[37] Lu Wei, Zhong Ma, Chaojie Yang, and Qin Yao. Advances in the Neural Network Quantization: A Comprehensive Review. *Applied Sciences*, 14(17):7445, August 2024.

[38] Xuechao Wei, Yun Liang, and Jason Cong. Overcoming Data Transfer Bottlenecks in FPGA-based DNN Accelerators via Layer Conscious Memory Management. In *Proceedings of the 56th Annual Design Automation Conference 2019*, pages 1–6, Las Vegas NV USA, June 2019. ACM.

[39] Ning Wu, Tao Jiang, Lei Zhang, Fang Zhou, and Fen Ge. A Reconfigurable Convolutional Neural Network-Accelerated Coprocessor Based on RISC-V Instruction Set. *Electronics*, 9(6):1005, June 2020.

[40] Xiang Yu, Zhijie Yang, Linghui Peng, Bo Lin, Wenjing Yang, and Lei Wang. CNN Specific ISA Extensions Based on RISC-V Processors. In *2022 5th International Conference on Circuits, Systems and Simulation (ICCSS)*, pages 116–120, Nanjing, China, May 2022. IEEE.

[41] Kh Shahriya Zaman, Mamun Bin Ibne Reaz, Sawal Hamid Md Ali, Ahmad Ashrif A Bakar, and Muhammad Enamul Hoque Chowdhury. Custom Hardware Architectures for Deep Learning on Portable Devices: A Review. *IEEE Transactions on Neural Networks and Learning Systems*, 33(11):6068–6088, November 2022.

[42] Bo Zhang, Shihui Yin, Minkyu Kim, Jyotishman Saikia, Soonwan Kwon, Sungmeen Myung, Hyunsoo Kim, Sang Joon Kim, Jae-Sun Seo, and Mingoo Seok. PIMCA: A Programmable In-Memory Computing Accelerator for Energy-Efficient DNN Inference. *IEEE Journal of Solid-State Circuits*, 58(5):1436–1449, May 2023.

[43] Yongan Zhang, Xiaofan Zhang, Pengfei Xu, Yang Zhao, Cong Hao, Deming Chen, and Yingyan Lin. AutoAI2C: An Automated Hardware Generator for DNN Acceleration on Both FPGA and ASIC. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 43(10):3143–3156, October 2024.