A Comparison of Neural Network Computation Architectures for Low-Power Edge Applications

Fabian Peddinghaus

***Abstract*—Neural networks (NNs) are widely used for machine learning (ML) applications, including speech recognition and object detection. While they deliver superb performance on many ML tasks, they come at the cost of high computational effort. Thus, deploying NNs on low-power edge devices is very challenging. A common solution to this problem involves real-time communication with external computing systems, such as cloud servers, to which sensor data is off-loaded for further processing. However, this increases latency, reduces reliability, and raises privacy concerns. Deploying NNs on specialized processing architectures at the edge of the network close to the data sources alleviates the aforementioned issues by performing computations locally without the need for a network connection. Despite their apparent advantages, edge-computing systems have to maintain energy efficiency and computational performance without reducing application accuracy or increasing hardware costs. Thus, numerous architectures have been devised and deployed in recent years, enabling the widespread adoption of NNs in intelligent edge systems.**

This paper provides a general overview of three computation architectures and their peculiarities to resolve the issue of efficient low-power NN inference at the edge. In particular, it presents a comparison of classical CPUs, vector processors, and specialized hardware accelerators.

I. Introduction

Resource-constrained edge devices, like smartwatches and smart speakers, have become ubiquitous in recent years. They commonly employ machine learning (ML) algorithms to cope with the large amounts of data that their sensors generate [1]. In particular, neural networks (NNs) have reached widespread adoption, allowing developers to provide intelligent edge systems with complex data analysis capabilities, such as gesture detection and keyword spotting. While NNs can detect data patterns that would be difficult to model with conventional algorithms, their high computational resource demand makes them challenging to implement. Many battery-powered ML applications are additionally limited to low power consumption, making computing efficiency a fundamental design objective for low-power ML systems [2].

The rapid advancements of ML algorithm research place additional pressure on developers to quickly incorporate newly introduced network features, requiring short hardware and software development times. While traditional CPUs are remarkably flexible due to their inherent generality, the slowdown of *Moores Law* has made them a suboptimal choice for power-constrained embedded systems [3]. Nevertheless, what remains generally constant in ML inference are parallelizable multiply-and-accumulate (MAC) operations, which form the basis of matrix-vector and matrix-matrix multiplications. Due to their inherent data-level parallelism (DLP), such operations lend themselves to acceleration through specialized computing paradigms, such as vector computations or hardware accelerators.

F. Peddinghaus is with the Department of Electrical and Computer Engineering, Technical University of Munich, 80333 Munich, Germany E-Mail: f.peddinghaus@tum.de.

While many different approaches have been taken to reduce the power consumption and increase the performance of deeply embedded ML systems, the here presented comparison concerns itself only with the high-level hardware architecture of computing systems. To that end, this paper analyzes and compares classical CPUs, vector processors, and specialized hardware accelerators with regard to their temporal and spatial properties, as well as their efficiency and flexibility when running NN workloads. It does not cover distinct microarchitectural or technology-specific optimizations like memristor arrays or other non-conventional arithmetic [1, 2, 4]. Furthermore, it does also not include model-specific optimization techniques, such as tensor decomposition, data quantization, or network sparsification [5, 6].

The paper is organized as follows: Section II presents related surveys that complement this work, while Section III introduces background information on neural networks. This is followed by an overview and comparison of computing architectures in Section IV and a summary in Section V.

II. Related Work

Numerous surveys have been conducted, each presenting existing neural network computing architectures, accelerator designs and implementations, as well as describing their techniques and comparing their performance. A good general overview of state-of-the-art academic and industrial ML hardware accelerators is provided in the 3-part survey series released by Reuther et al. in 2019 [7], 2020 [8], and 2021 [9].

A comprehensive overview of tools, methods, and architectures for low-power NN accelerators can be found in [2]. A similarly rigorous survey with a general view on software design patterns, loop scheduling and optimization paradigms from application software to hardware-level data flow is presented in [5]. Deng et al. [6], as well as [10], provide a more optimization-focused approach emphasizing model optimization and compression techniques in conjunction with hardware architectures.

Sze et al. [1] provide a detailed survey on the efficient processing of NNs, incorporating historical aspects, common network models, training frameworks, and popular datasets. In 2020 Sze and her colleagues released their work as a book in

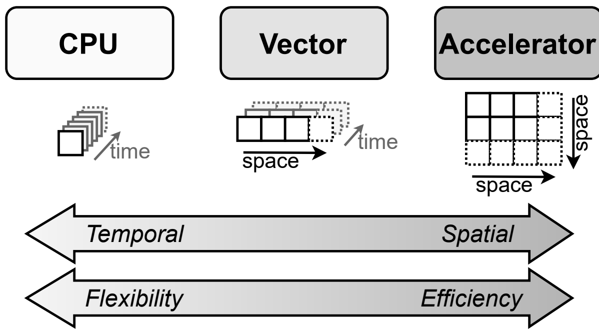

Fig. 1: Taxonomy of different computation architectures with respect to spatial and temporal properties.

the Synthesis Lectures on Computer Architecture series [3]. It provides a comprehensive introduction and broad overview of machine learning computer architectures, covering numerous low-power and edge computing aspects.

III. Background on Neural Networks

The fundamental component of a neural network (NN) is the neuron. For each neuron, the inputs are multiplied with the neuron's weights and added together with a bias term. A non-linear activation function is then applied to compute the neuron's output. In fully connected neural networks (FCNNs), every neuron in a layer is connected to all its predecessors. The weights of each neuron are trained by using vast amounts of training data and a method called back-propagation. In back-propagation, the detection error of the output is propagated backwards through the network, and every weight is adjusted proportionately to its relative share of the output error [3, 6].

Although FCNNs can be used to learn features and classify data, their high degree of connectivity makes them impractical for larger inputs such as high-resolution images. Moreover, FCNNs treat inputs that are close together or far apart equivalently, ignoring the spatial structure present in images [5]. To overcome these limitations, convolutional neural networks (CNNs) exploit the idea of local receptive fields and shared weights, reducing the number of free parameters and thus enabling deeper networks. CNNs employ so-called kernels, which they slide across the input matrix of the layer. Pooling layers help reduce the number of activations of a layer and consequently decrease the memory requirements and the number of computations to be performed [3, 5].

Inference of both FCNNs and CNNs can generally be reduced to simple vector-matrix or matrix-matrix multiplications. However, while FCNNs and CNNs comprise a large majority of current ML workloads, many more NN architectures exist. Furthermore, most NNs employ additional layers and activations, which are generally more complex to compute [3, 5, 6]. Because NNs are a prevalent research topic, novel architectures are constantly being introduced, further diversifying the landscape of NN models [11]. Thus, NN computation systems require a certain degree of generality and flexibility in order to be able to compute the large and constantly evolving volume of NN models [12, 13].

IV. Comparison

This section provides a comparison of three distinct architectures for efficient machine learning at the edge. It presents and compares traditional CPUs, vector machines, and specialized ML accelerators. As already pointed out, this section focuses on providing a high-level overview of each architecture by detailing its strengths and weaknesses. Additionally, it provides examples of real-world implementations from academia and industry. Before diving into the actual comparison, the following subsection briefly introduces the difference between spatial and temporal architectures in accordance with Figure 1.

A. Temporal vs. Spatial

A fundamental property of temporal architectures is that they feature a central control unit that schedules instructions and distributes computations across one or many processing elements (PEs). Temporal designs are commonly based on load-store architectures, in which data is moved between the physically separated computational and memory units. Temporal architectures typically operate on scalar values and are generally characterized through a single instruction, single data (SISD) processing scheme [2]. In order to share resources, temporal architectures employ time-multiplexing of the execution. While such an approach reduces the resource requirements, it leads to an increase of overall execution time [14]. The most prominent embodiment of a temporal architecture is the widespread *von Neumann* based central processing unit (CPU). Moreover, some authors argue that the traditional vector architecture, with its single instruction multiple data (SIMD) scheme, can also be categorized as a temporal architecture [2, 15].

In contrast to temporal architectures, spatial architectures allow their PEs to move data between adjacent PEs. They reduce memory accesses and overall data traffic by employing local buffers. Systolic arrays (see Figure 2) are the most prominent representatives of such a spatial computation pattern in the realm of ML accelerators. They enable efficient data reuse by employing a tightly coupled 2D architecture [2]. Such architectures can be efficiently utilized when implementing matrix multiplication through spatial concatenation of computations, thus minimizing memory accesses [14].

The classification of temporal and spatial architectures can be ambiguous, depending on the viewpoint or abstraction level. It should not be seen as a binary categorization but rather as a continuous mapping (see Figure 1).

B. CPU

Employing low-power CPU cores for ML and NN workloads results in low-area, low-power, and low-cost computation systems, making it possible to deploy them in a wide range of applications. This gave rise to TinyML [16], which consists of running NN models on extremely resource-constrained microcontrollers with a power consumption of less than 1mW. The authors from [17] argue that edge devices that have to cope with highly time-varying workloads, characterized by

bursts of compute-intensive operations amid long periods of low activity, as common in many ML edge applications, can significantly benefit from highly-efficient CPU cores. Additionally, they suggest that architectural heterogeneity inside these cores could potentially provide a possible solution to harmonize competing optimization goals. They present two ultra-low-power RISC-V cores, each consuming significantly less than 1mW and run a number of ML and NN benchmarks on them. These systems are so energy efficient that they could theoretically run for months on a single coin battery.

ARM provides a similar lineup of ultra-low-power cores through their Cortex-M0 series microcontrollers [18]. Akin to the RISC-V cores, they can run ML workloads while consuming only minimal amounts of energy and chip area. The whole CPU core can be synthesized using not more than 15KGE [17], making them highly cost-effective. Together with machine learning libraries, like CMSIS-NN [19] and the TensorFlow Lite for Microcontrollers (TFLM) software framework [20], these processors provide a very attractive solution for deeply embedded systems and ultra-low-power edge devices.

C. Vector Architectures

Instead of operating on only a single data value at a time, vector architectures simultaneously operate on an array of data values according to the SIMD pattern. In contrast to manycore architectures or GPUs, all execution units are synchronized and respond to a single instruction issued from a single program counter (PC). Thereby, they efficiently exploit the inherent data-level parallelism (DLP) of ML workloads. Through vector instructions, the compiler is able to expose DLP to the hardware, thus reducing the complexity otherwise encountered in out-of-order machines and enabling more efficient parallel compute. Another key advantage of vector architectures is that they amortize the cost of control logic over many execution units while simultaneously reducing the required instruction memory and bandwidth.

Vector processors utilize multiple parallel computation pipelines, commonly known as vector lanes, producing two or more results simultaneously [13]. A prominent example of such a vector architecture is the ARM Neon extension. It can be used to accelerate both FCNNs and CNNs, resulting in a speedup of more than 3.5x in execution time and an energy improvement of almost 10x when using eight 16-bit vector lanes [21]. ARMs CMISIS-NN ML library makes use of Helium, a vector extension for the area constrained M-Cortex series [19]. It provides optimized scalar implementations, as well as vectorized kernels, and offers integration with TFLM [20].

There are also a number of academic vector processors available that support the RISC-V vector extension. An exciting RISC-V vector core for low-power edge applications is a processor designed by researchers from the University of Southhampton [22]. The authors developed a synthesizable vector processor consisting of four 8-bit vector lanes, resulting in resource utilization of no more than 2.6x when compared to a scalar CPU core. Substantial performance gains were

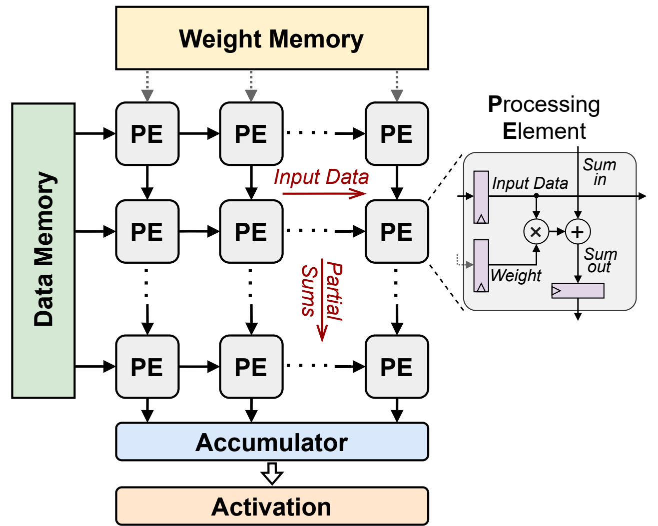

Fig. 2: High level diagram of a weight-stationary systolic array architecture (derived from [23]).

observed despite its small size, illustrating the advantages of vector processing in low-power microcontrollers. When put next to a scalar baseline implementation, measured runtime improvements range from 2.7x to 5.8x.

Compared to specialized accelerators, a significant advantage of vector architectures is their generality and programmability. From a software perspective, they behave very similar to traditional scalar CPUs and can thus be programmed more effectively than specialized accelerators. While most ML workloads consist of multiply-accumulate (MAC) operations, modern NNs quickly evolve beyond traditional feed-forward architectures. A prominent example are language models, which make heavy use of the so-called transformer architecture [11]. Such diverse workloads cannot easily be mapped onto specialized accelerators. Hence, frequent data moved between the accelerator and host CPU is required, which further strains the already limited bandwidth and results in poor overall performance. Using the flexibility of vector architectures, one can more easily accelerate state-of-the-art models as they emerge from ML research while still benefiting from the parallel and efficient compute offered by the vector computation paradigm.

D. Specialized Accelerators

This section details specialized hardware architectures for NN workloads. The presented designs follow the already mentioned spatial computation pattern that aims to improve throughput and reduce energy consumption by intrinsically exploiting DLP in hardware and reducing internal data movement. Through the reuse of local data, such as network weights and feature maps, these designs are able to minimize the traffic of input data and partial sum accumulations. Such dataflow patterns can be found in many prominent accelerators, such as the Google Edge TPU [24] and the Eyeriss accelerator [25].

Figure 2 depicts a block diagram of a typical NN accelerator modeled closely to that of the Edge TPU [23]. It is based on a weight-stationary systolic array architecture, encompassing a grid of PEs that accelerates matrix multiplications. When

computing a batch of input data for a single FCNN layer, weights are pre-loaded in a transposed manner from the top into the 2D array of PEs. They are stored in local buffers inside the PEs and remain stationary throughout the computation. The PEs contain simple MAC units that feature multipliers and adders. Next, data values stream in from the data memory and propagate from left to right through the systolic array. The partial sums of the output matrix are computed and proceed downwards through the grid into the accumulators, where a bias term is added. Following the completion of the matrix multiplication, an activation function is applied to each result of the output matrix in the accumulator [23]. While CNN layers can generally use the same computation architecture by loading weights and data values into the systolic array with a slightly modified schedule, the authors from [25] present a more sophisticated architecture directly tailored towards inference of CNNs.

Modern edge NN workloads exhibit high diversity in terms of data reuse. This heterogeneity necessitates the need for flexible dataflow patterns in computation architectures. However, state-of-the-art accelerators such as the Edge TPU and Eyeriss implement a single dataflow pattern designed for high spatial reuse of a specific NN type. Thus, they are unable to accommodate varying workloads. The authors from [26] claim that the Edge TPU actually suffers from extreme underutilization of its PEs, resulting in poor energy efficiency. Albeit consuming large amounts of dynamic power, the Edge TPU's overprovisioned on-chip buffers fail at effectively caching parameters for more than a single layer due to the inherent diversity of NN workloads. This causes more than half of the inference energy to be consumed by parameter communication and off-chip data movement. Moreover, according to the authors, the Edge TPU spends approximately three-quarters of its total energy on DRAM accesses when computing diverse NN workloads. While the buffers consume a significant amount of area in the Edge TPU, they are ineffective at reducing off-chip memory accesses, amortizing any gains achieved by its highly optimized systolic array.

V. Conclusion

This paper provides a comparison of efficient low power NN architectures for ML edge applications. While the presented CPU based systems offer energy-efficient inference, they are unable to provide the required performance of ML workloads. Specialized accelerators deliver superb efficiency and performance but are unable to encompass the diversity of contemporary NN models. These shortcomings are not unique to the above presented Edge TPU but rather a systemic problem of specialized hardware accelerators, particularly in the embedded space where memories and buffers are small, and workloads are diverse. This requires further research throughout the whole stack, from software frameworks to hardware design. It is inevitable and necessary to rethink computation architectures for NN edge applications. The outlined vector approach attempts to balance computational efficiency with generality, promising excellent potential for further improvements.

References

[1] Vivienne Sze et al. "Efficient processing of deep neural networks: A tutorial and survey". In: *Proceedings of IEEE* (2017).

[2] Petar Jokic et al. "A Construction Kit for Efficient Low Power Neural Network Accelerator Designs". In: *arXiv preprint arXiv:2106.12810* (2021).

[3] Vivienne Sze et al. "Efficient processing of deep neural networks". In: *Lectures on Computer Architecture* (2020).

[4] Kyuho Jason Lee et al. "The Development of Silicon for AI: Different Design Approaches". In: *IEEE Transactions on Circuits and Systems* (2020), pp. 4719–4732.

[5] Maurizio Capra et al. "Hardware and software optimizations for accelerating deep neural networks: Survey of current trends, challenges, and the road ahead". In: *IEEE* (2020).

[6] Lei Deng et al. "Model compression and hardware acceleration for neural networks: A comprehensive survey". In: *Proceedings of the IEEE* 108.4 (2020), pp. 485–532.

[7] Albert Reuther et al. "Survey and benchmarking of machine learning accelerators". In: *2019 IEEE HPEC*. 2019, pp. 1–9.

[8] Albert Reuther et al. "Survey of machine learning accelerators". In: *2020 IEEE HPEC*. 2020, pp. 1–12.

[9] Albert Reuther et al. "AI Accelerator Survey and Trends". In: *arXiv preprint arXiv:2109.08957* (2021).

[10] Lukas Baischer, Matthias Wess, and Nima TaheriNejad. "Learning on Hardware: A Tutorial on Neural Network Accelerators and Co-Processors". In: *arXiv preprint* (2021).

[11] Ashish Vaswani et al. "Attention is all you need". In: *NeurIPS*. 2017, pp. 5998–6008.

[12] Jian Weng Liu and Tony Nowatzki. "Generality is the Key Dimension in Accelerator Design". In: *Power* (2021).

[13] Yunsup Lee et al. "Exploring the tradeoffs between programmability and efficiency in data-parallel accelerators". In: *ISCA*. 2011, pp. 129–140.

[14] S. Takano. *Thinking Machines: Machine Learning and Its Hardware Implementation*. Elsevier Science, 2021.

[15] Maurizio Capra et al. "An updated survey of efficient hardware architectures for accelerating deep CNNs". In: (2020).

[16] P. Warden and D. Situnayake. *TinyML: Machine Learning with TensorFlow Lite on Arduino and Ultra-low-power Microcontrollers*. O'Reilly, 2020.

[17] Pasquale Davide Schiavone et al. "Slow and steady wins the race? A comparison of ultra-low-power RISC-V cores for Internet-of-Things applications". In: *PATMOS*. IEEE. 2017.

[18] ARM Limited. "ARM Cortex M0 Technical Reference Manual". In: Revision C (2009).

[19] Liangzhen Lai, Naveen Suda, and Vikas Chandra. "Cmsis-nn: Efficient neural network kernels for arm cortex-m cpus". In: *arXiv preprint arXiv:1801.06601* (2018).

[20] Martin Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. 2015.

[21] Sung-Jin Lee, Sang-Soo Park, and Ki-Seok Chung. "Efficient SIMD implementation for accelerating convolutional neural network". In: *ICCIP*. 2018, pp. 174–179.

[22] Matthew Johns and Tom J Kazmierski. "A Minimal RISC-V Vector Processor for Embedded Systems". In: *2020 FDL*. IEEE. 2020.

[23] Jeff Zhang et al. "Thundervolt: enabling aggressive voltage underscaling and timing error resilience for energy efficient deep learning accelerators". In: *DAC*. 2018.

[24] Amir Yazdanbakhsh et al. "An evaluation of edge tpu accelerators for convolutional neural networks". In: *arXiv preprint arXiv:2102.10423* (2021).

[25] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks". In: *ACM SIGARCH* (2016), pp. 367–379.

[26] Amirali Boroumand et al. "Google Neural Network Models for Edge Devices: Analyzing and Mitigating Machine Learning Inference Bottlenecks". In: *PACT*. IEEE. 2021, pp. 159–172.